Pattern Matching and Applications

by

Mohammad Alimuddin

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

ELECTRICAL ENGINEERING

October, 1990

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

PATTERN MATCHING AND APPLICATIONS

BY

MOHAMMAD ALIMUDDIN

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

ELECTRICAL ENGINEERING

OCTOBER 1990

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by ***MOHAMMAD ALIMUDDIN*** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of *MASTER OF SCIENCE* in *ELECTRICAL ENGINEERING.*

THESIS COMMITTEE

Dr. G. F. Beckhoff (Chairman)

Dr. Al-Akhdhar, Sayed Zaki (Member)

Dr. M. Atiquzzaman (Member)

Electrical Engineering Department Office of the Chairman

Department Chairman

Dean, College of Graduate Studies

كلية الدراسات العليا COLLEGE OF GRADUATE STUDIES

Date: 30th October, 1990

To My Beloved Parents, Brothers

and to Those Who Shared Their Care and Concern

ACKNOWLEDGMENT

All praises and glory be to Allah for giving me the knowledge, wisdom and patience to accomplish this research work.

Acknowledgement is due to *King Fahd University of Petroleum and Minerals* for support of this research.

I express my deep gratitude to *Prof. Gerhard F. Beckhoff*, my thesis advisor, for his patient guidance and generous support during the course of this research.

I also thank my thesis committee members, *Dr. Al-Akhdhar*, *Sayed Zaki* and *Dr. Mohammad Atiquzzaman* for their valuable suggestions and their co-operation.

I owe sincere thanks to my colleagues and friends, *Siyad Ma*, *Asjad Mumtaz*, *Azhar Sayed*, *Nizamuddin*, *Saleem* and *Shahid* for their moral support and assistance during the course of my research work. I also thank my other colleagues *Aleem Paracha*, *Azhar Rana* and *Wasim Akhtar* and all my *Brothers*, *Friends* for their encouragement and support.

Special thanks are due to *Mr. Hanifa M. Nasir* for allowing me to use his computer and his expertise help in using Turbo-Pascal.

I am grateful to *Mr. Ahmed Sheikh* and *Mr. Mutlaq* of the Computer Graphics Centre for their help in using the graphic facilities.

ملخص

العنوان : موافقة النماذج وتطبيقاتها

تأليف : محمد عليم الدين

التخصص : هندسة كهربائية

التاريخ : ربيع الأول ١٤١١هـ أكتوبر ١٩٩٠م

عند ربط موافقة النماذج بموافقة سلاسل الرموز فإن هذا الموضوع يظهر غنياً بالتطبيقات . وتعد عملية ضغط سلسلة من الرموز إحدى أهم المسائل التى يتكرر مواجهتها فى موافقة سلاسل الرموز . وتتمثل عملية الضغط هذه فى إيجاد أقصر سلسلة رموز ممكنة بحيث تحتوى على جميع السلاسل الموجودة فى مجموعة ما كسلاسل جزئية منها . تسمى هذه السلسلة المضغوطة السلسلة الكلية المشتركة الصغرى (س . ك . م . ص.) .

تم دراسة خوارزميات لتطوير الـ (س.ك.م.ص.). ولأنه لا توجد خوارزمية كثيرة حدود لحل هذه المسألة ، فقد تم دراسة خوارزميات كثيرات حدود تقريبية . هذا وقد قدمنا خوارزمية الرسم البيانى المختصر والتى تعتمد على حلول تجريبية ويمكن تنفيذها بنائياً .

تم استخدام خوارزمية الـ(س.ك.م.ص.) المطورة هنا لأنتاج متواليات فحص تسمى تجارب الإختبار للآلات التتابعية . وقد ثبت أن طريقة المتوالية الكبرى تنتج تجارب اختبار أقصر من تلك المنتجة بالطريقة التقليدية المعتمدة على طريقة « هينى » .

ختاماً ، وكخطوة أولى نحو حل بنائى لـ (س.ك.م.ص.) فقد تم تصميم دائرة تكثيف عال جداً خاصة لأيجاد المقاطع المتشابهة بين السلاسل ، وكان تصميم دائرة التكثيف على شكل منظومة ضخ . كما تم تركيب دائرة التكثيف العالى جداً من الخلايا الأولية .

ماجستير العلوم

جامعة الملك فهد للبترول والمعادن

الظهران - المملكة العربية السعودية

ربيع الأول ١٤١١هـ أكتوبر ١٩٩٠م

ABSTRACT

Title : **PATTERN MATCHING AND APPLICATIONS**

By : **Mohammad Alimuddin**

Major Field : **Electrical Engineering**

Date : **October 1990**

Pattern matching as related to string matching finds extensive application. One of the most interesting problems often encountered in string matching is that of string compression, i.e., to find the shortest possible string that contains every string in a given set as substrings. Such a shortest possible string is called a SHORTEST COMMON SUPERSTRING(SCS).

Algorithms for developing the SCS have been investigated. The SCS problem is NP-complete, meaning that there is no polynomial-time algorithm to solve this problem. Hence polynomial-time approximation algorithms have been investigated. We have proposed the Graph-Reduction Algorithm which lends itself to heuristic solutions and could possibly be used for implementation by hardware.

The Graph-Reduction Algorithm proposed for SCS has been applied to the generation of test sequences called checking experiments for sequential machines. The SCS method referred to as the Supersequence Construction Method is shown to yield shorter length checking experiments than the conventional method based on Hennie's procedure.

Finally as a first step towards a hardware solution for the SCS, an efficient implementation of a **special-purpose VLSI chip** has been given to find **overlaps** between strings. The chip has been designed as a systolic array. A VLSI implementation of the basic cells has been carried out in CMOS.

MASTER OF SCIENCE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

October 1990

TABLE OF CONTENTS

Chapter *Page*

LIST OF TABLES

LIST OF FIGURES

LIST OF NOTATIONS

Σ *finite alphabet set*

Σ^* *set of finite sequences* over Σ

lg *length of*

$\bullet$ *overlapping operator*

ψ *overlap length*

G *graph*

V *finite set of vertices*

E *finite set of edges*

H *Hamiltonian path constructed by approximation algorithm*

H_{max} *longest Hamiltonian path (optimal)*

wt *weight function*

s_p *optimal length superstring*

s_g *superstring constructed by Greedy Algorithm*

M *finite state machine*

Q *state set*

A *input alphabet*

Y *output alphabet*

δ *next-state function*

λ *output function*

$\times$ *cartesian product*

A^* *set of all input sequences*

Y^* *set of all output sequences*

e *the empty (null) sequence*

$\in$ *is a member of*

ν *response pair function*

δ_w *next-state function under input w, state equivalence under w*

λ_w *output function under input w, output equivalence under w*

ν_w *response pair function under input w, state-output equivalence under w*

$\hat{\delta}$ *previous state function*

$P(Q)$ *set of all partitions on a set Q*

Φ *the identity relation*

ω *the universal relation*

Λ *meet*

$\exists$ *there exists*

$\forall$ *for all*

$\neq$ *not equal to*

$<=>$ *if and only* if

$|w|$ *length of sequence w*

$=>$ *implies*

$\cup$ *union*

$|A|$ *number of inputs*

$|Q|$ *number of states*

$\tilde{\delta}$ *input-state function*

d *D-sequence*

$\leq$ *less than or equal to*

s *input filtering function*

CHAPTER 1

INTRODUCTION

1.1 GENERAL INTRODUCTION

Pattern matching problems have always posed an interesting problem to researchers. Pattern matching as related to string matching finds application in practically every field, particularly in the area of digital technology.

String matching is an essential part of information transmission and coding theory. In coding of information, variable length codes are often necessary to reduce the average length of the coded message. For a uniquely decipherable code the decoding procedure involves matching the received string with legitimate code strings from both ends and the code assignment which is consistent from both ends gives the decoded output [1]. A similar procedure is used in input retrieval for information lossless sequential machines [1]. A machine is said to be information lossless if the knowledge of the initial state, the output sequence and the final state is sufficient to determine uniquely the input sequence [1].

In many information retrieval and text editing applications it is necessary to be able to quickly locate some of the occurrences of user specified patterns and phrases in text. To locate all occurrences of a finite number of keywords and phrases in an arbitrary text string, efficient string matching is necessary [2].

In VLSI it is required that a design layout conforms to the design rules laid down by the particular technology [3]. A design rule checker is often a piece of software which is based on checking of patterns in ratios of 1's and 0's, the 1's and 0's being respectively the pixels of a layer in the design layout [3].

Digital testing is another important application where matching bit patterns is essential. Checking experiments involve catenation of certain sequences. In order to minimize the length of the checking experiments an optimal overlap needs to be found between these sequences. Other applications include molecular biology and data compression [4].

String compression is one of the interesting problems often encountered in string matching applications; i.e., given a set of strings to find the shortest possible string that contains every string in the set as a substring. Such a shortest possible string is called as 'Shortest Common Superstring' [4]. The work reported in this thesis relates to the Shortest Common Superstring (SCS) problem. In the following sections to follow, the reasons for our interest in the Shortest Common Superstring (SCS) problem will be enumerated. A short literature review is presented, followed by proposed work and thesis organization.

1.2 LITERATURE REVIEW

The earliest work on the SCS problem was done by Gallant [5]. Gallant showed that the SCS problem was NP-complete, meaning that there is no

polynomial-time algorithm to solve this problem. He used a simple greedy algorithm to obtain SCS. Storer and Szymanski [6], have used the SCS problem for data compression. Tarhio and Ukkonen [4] have dealt with the SCS problem for recovering DNA sequencing information from experimental data. They have analyzed the performance of a greedy approximation algorithm in developing SCS. Turner [7] has given several algorithms for the SCS problem, which include a matching algorithm, dimatching algorithm and an efficient implementation of the greedy algorithm. The application area addressed by Turner also has been data compression and DNA sequencing.

In the application of digital testing, traditionally checking experiments for fault diagnosis of sequential machines have been developed using Hennie's [1] procedure. Recently a method for developing shorter length checking experiments has been proposed in [8]. This method can be modelled as a SCS problem, and is the main motivation for this research work.

The SCS problem has been viewed as a longest Hamiltonian path problem in complete weighted directed graphs [4,7]. This requires the finding of overlaps between the given set of strings. There is an efficient pattern matching algorithm [9] which can be used to find overlaps [4]. Also there have been hardware implementations to find a pattern in a given string [10,11]. But there has been no hardware implementation to find overlaps between strings. An efficient implementation to find overlaps by hardware is discussed in Chapter 4.

1.3 THE PROPOSED WORK

The SCS problem is investigated. The overlaps required to form a SCS are found using a modification of the Knuth-Morris-Pratt (KMP) algorithm. Since a software algorithm is likely to be slow, a hardware solution is investigated. A preliminary finite input memory machine implementation is given. Being pattern dependent, this design is not extendable. Hence an efficient implementation of a special-purpose VLSI chip based on a systolic array design is proposed.

The graph-reduction algorithm for finding a SCS from the overlapping information is proposed. This algorithm lends itself to heuristic solutions, and could possibly be implemented in hardware.

The SCS problem has been applied in developing shorter length checking experiments by using the essential sequences [8] as the string set. It is shown by illustrative examples that the SCS method, which we call the Supersequence Construction Method, yields shorter length checking experiments compared to conventional Hennie's [1] procedure.

1.4 THESIS ORGANIZATION

In chapter 2, the overlaps and Shortest Common Superstring (SCS) are formally defined. The steps to find SCS are enumerated. The KMP algorithm for pattern matching is reviewed. This is followed by a preliminary hardware design based on a finite input memory machine to find overlaps. However, because of pattern dependence, this design is not very efficient. Therefore, we

propose an efficient systolic design of a special-purpose VLSI chip to find overlaps. The discussion of the design is however deferred until Chapter 4. The greedy algorithm and matching algorithm for SCS are reviewed. At the end of this chapter we propose the graph-reduction algorithm for SCS. An illustrative example is given to show the use of heuristics in this algorithm.

Chapter 3 deals with development of shorter length checking experiments, by applying the SCS method to a string set of essential sequences [8]. Preliminary definitions required for state identification of sequential machines are given. Two methods to find D-sequences are reviewed. The Supersequence Construction Method based on developing SCS of essential sequences by the graph-reduction algorithm is compared with the conventional Hennie's procedure, by illustrative examples.

An efficient systolic design of a special-purpose VLSI chip to find overlaps is proposed in Chapter 4. A brief review of systolic arrays is presented at first. The working of the special-purpose VLSI chip, which we call the **Overlap Chip** , is illustrated. This is followed by a discussion of the data-flow algorithm. Then the basic cells required in the systolic array are discussed. The cell algorithms and cell circuits are discussed in detail. Finally, the cell layouts based on the CMOS VLSI design rules in [12] are presented.

Finally, in Chapter 5, conclusions and suggestions for future work are given. The Pascal programs for finding maximal overlaps using a modification of KMP, for finding SCS by greedy algorithm and graph-reduction algorithm, and

and for finding minimal length D-sequences by both the sets and partitions method are given in the appendix.

CHAPTER 2

THE SHORTEST COMMON SUPERSTRING:

2.1 INTRODUCTION

The Shortest Common Superstring problem (SCS) is to find a shortest possible string that contains every string in a given set as substrings. Let $s_1 = a_1a_2 \ldots a_r$ and $s_2 = b_1b_2 \ldots b_s$ be strings over some finite alphabet Σ, where $a_i \in \Sigma$, $1 \le i \le r$ and $b_j \in \Sigma$, $1 \le j \le s$. Σ represents the alphabet (consisting of letters or symbols) over which the strings are written. Let Σ^* represent the infinite set of finite length sequences of letters or symbols over Σ including the empty string e of length zero. We define Σ^+ as $\Sigma^* - \{e\}$. The length of a string $s_1 \in \Sigma^*$ is defined as the number of letters or symbols in that string and is denoted by $lg(s_1)$. We say that s_1 is a substring of s_2 or s_2 is a superstring of s_1 if $\exists\, x,y \in \Sigma^*$ such that $xs_1y = s_2$ where $s_1, s_2 \in \Sigma^+$. The shortest common superstring problem (SCS) can now be stated as: Given a finite non-empty set of strings $S = \{s_1, s_2, \ldots s_n\}$ where $s_1, s_2, \ldots, s_n \in \Sigma^+$, find a minimal length string s_p that is a superstring of every $s_i \in S$.

A set of strings S is said to be substring free if no string in the set S is a substring of any other. In our discussion here we assume that the sets we deal with are substring-free. This involves no loss of generality, since any set of strings from which substrings have been removed has the same SCS as the

original set. In fact for the application presented in Chapter 3 all the string sets are substring-free sets, since all strings are different and have the same length.

The viewpoint of the SCS problem we have taken so far, is to minimize the length of common superstrings. An alternative viewpoint leading to the same result is that of finding an ordering of the strings that maximizes the amount of overlap between consecutive strings [4]. Let us investigate this viewpoint further.

Let $s_1 = a_1a_2 \ldots a_r$ and $s_2 = b_1b_2 \ldots b_s$ be strings, $s_1, s_2 \in \Sigma^+$. We define the maximal overlap $\psi(s_1,s_2) = \max\{k \geq 0 \mid a_{r-k+i} = b_i,\ 1 \leq i \leq k\}$. If $\psi(s_1,s_2) = k$ then $s_1 \bullet s_2$ is defined to be the string $a_1a_2 \ldots a_r b_{k+1} \ldots b_s$. We note that if s_1,s_2,s_3 are strings, none of which is a substring of another, then $s_1 \bullet (s_2 \bullet s_3) = (s_1 \bullet s_2) \bullet s_3$ i.e., the overlapping operation is associative.

Let $S = \{s_1,s_2, \ldots ,s_n\}$, where $s_1,s_2, \ldots ,s_n \in \Sigma^+$. Let $s_1s_2 \ldots s_n$ represent the catenation of strings $s_1,s_2, \ldots$ and s_n. Then there are n! permutations associated with $s_1s_2 \ldots s_n$. Let $s_{\pi_1}s_{\pi_2} \ldots s_{\pi_n}$ be one such permutation such that the sum of overlaps $\psi_\pi = \sum_{i=1}^{n-1} \psi\{s_{\pi_i} ,s_{\pi_{i+1}}\}$ is maximum. Then the minimal length superstring is $s_p = s_{\pi_1} \bullet s_{\pi_2} \ldots \bullet s_{\pi_n}$.

Example 2.1:

Let $S = \{s_1,s_2,s_3,s_4,s_5\} = \{hfdegi, bfgiak, iakhfd, egiach, fgiakh\}$.

Then $s_{\pi_1}s_{\pi_2}s_{\pi_3}\ s_{\pi_4}s_{\pi_5} = s_2s_5s_3s_1s_4$

$= hfgiakfgiakhiakhfdhfdegiegiach$

and the sum of overlaps

$\psi_\pi = \psi(s_{\pi_1},s_{\pi_2}) + \psi(s_{\pi_2},s_{\pi_3}) + \psi(s_{\pi_3},s_{\pi_4}) + \psi(s_{\pi_4},s_{\pi_5})$

$= \psi(s_2,s_5) + \psi(s_5,s_3) + \psi(s_3,s_1) + \psi(s_1,s_4)$

$= 5+4+3+3 = 15.$

The superstring, $s_p = s_{\pi_1} \bullet s_{\pi_2} \bullet s_{\pi_3} \bullet s_{\pi_4} \bullet s_{\pi_5} = s_2 \bullet s_5 \bullet s_3 \bullet s_1 \bullet s_4 = hfgiakhfdegiach$

and $lg(s_p) = 15.$

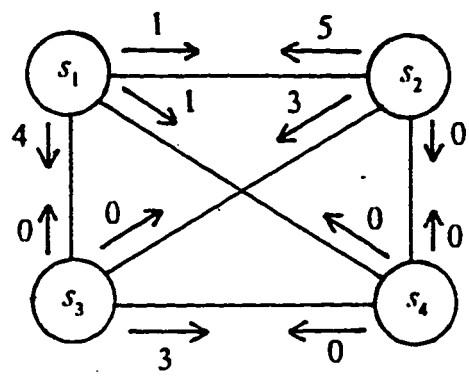
Hence we can view the object of the SCS problem as that of finding an ordering of the strings in S such that the sum of overlaps ψ_π is maximum. This can now be viewed as a special case of the longest Hamiltonian path problem for weighted directed graphs [4] and will be dealt with in the next section. Formally we define the following : A *graph*, G is a pair (V,E) where V is a finite set and E is a subset of $V \times V$. Elements of V are called vertices and elements of E are called edges. If the edges of a graph are identified with ordered pairs of vertices, then G is called a *directed graph.* A *weighted directed graph* is a triple (V,E, *wt*) where (V,E) is a directed graph and *wt* is a function from E into Z, the set of integers. If $(u,v) \in E$, then $wt((u,v))$ is called the weight of the edge (u,v). A *complete graph* G is a graph with an edge between each pair of distinct vertices. If a complete graph G has n vertices, then it has $\frac{n(n-1)}{2}$ edges.

The subsequent section will deal with the construction of the overlap graph and steps to find SCS. A straight forward pattern matching algorithm and the KMP algorithm are then discussed. It is shown, how the KMP algorithm is modified to find overlaps. Since software implementations are likely to be slow, a hardware implementation based on a finite input memory machine is presented. This design is not easily extendable since it is pattern dependent. A more efficient hardware implementation is discussed in Chapter 4. Finally the greedy algorithm and matching algorithm for SCS are reviewed and the proposed graph-reduction algorithm is illustrated with examples.

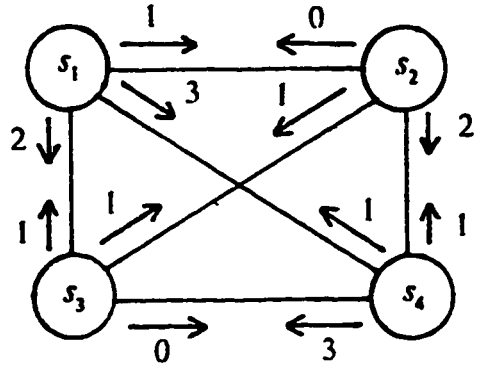
2.2 LONGEST HAMILTONIAN PATH PROBLEM

In this section we look into the construction of the weighted directed graph which will lead us to the solution of the SCS problem. Let S = $\{s_1, s_2, \ldots, s_n\}$ be a substring free set. The overlap graph for S is defined as follows: The graph is a complete weighted directed graph with vertex set V = $\{s_1, s_2, \ldots, s_n\}$. Each edge $(s_i, s_j) \in E$, has a weight equal to the maximal overlap $\psi(s_i, s_j)$, i.e., $wt((s_i, s_j)) = \psi(s_i, s_j)$ where $1 \le i, j \le n$.

Example 2.2 : S = $\{s_1, s_2, s_3, s_4\}$ = { cbadef, fcbade, adefcd, fcdafb]



Example 2.3 : $\{s_1,s_2,s_3,s_4\} = \{$ 1001,1000,0111,0011$\}$



A path p in a graph is defined as a sequence of adjacent vertices (i.e., having an arc directly connecting them). A Hamiltonian path in a graph is a path which includes every vertex in that graph. Let p be any path in G. The weight of path p in G is defined to be the sum of the weights of its edges and is denoted by $wt(p)$. In order to maximize the sum of overlaps $\psi_\pi = \sum_{i=1}^{n-1} \psi(s_{\pi_i}, s_{\pi_{i+1}})$ we need to find a Hamiltonian path in the overlap graph constructed, such that its weight is maximum. We denote such a longest Hamiltonian path as H.

The steps involved in finding the shortest common superstring can be summarized as follows:

1. Compute the maximal pairwise overlaps $\psi(s_i,s_j)$ between all strings in $S = \{s_1,s_2,\ldots,s_n\}$ for $1 \le i,j \le n$ and $i \neq j$.
2. Construct the overlap graph for S.
3. Find a longest Hamiltonian path H in the overlap graph. If $s_{\pi_1}, s_{\pi_2}, \ldots, s_{\pi_n}$ are the elements of S written in the order they appear on the path H then the shortest common superstring is $s_{\pi_1} \bullet s_{\pi_2} \ldots \bullet s_{\pi_n}$.

In the subsequent sections we consider the above steps in detail. We also develop several algorithms which will lead us to the desired result.

2.3 OVERLAPPING

The problem to be solved here is the following: Given two strings s_1 and s_2, determine the maximal overlap $\psi(s_1,s_2)$ between s_1 and s_2.

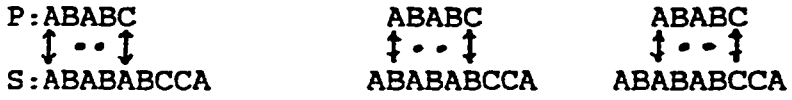
This problem can be understood as a generalization of the classical pattern-matching problem, where s_1 is the text and s_2 is the pattern and we ask whether the pattern occurs in the text. For the overlapping problem we also ask what is the longest suffix of the text which is also a prefix of the pattern.

2.3.1 Straight-Forward Algorithm

A very straightforward algorithm [13] for the solution of the classical pattern matching problem would proceed as follows. Starting at the beginning of each string, we compare characters, one after the other, until either the pattern is exhausted or a nonmatch is found. In the former case we are done; a copy of the pattern is found in the text. In the latter case we start over again, comparing the first pattern character with the second text character. Whenever a nonmatch is found, we move the pattern one more place forward in the text and start again comparing the first pattern character with the next text character.

Example 2.4 : Comparisons are done (from left to right) on the pairs of

characters indicated by arrows:



Observe that moving the pattern all the way past the point where the mismatch occured could fail to detect an occurrence of the pattern.

Algorithm : Straightforward String-Matching Algorithm.

Input : P and S, the pattern and text strings, both non-null.

Output : A success or failure indicator. Also location, i, of pattern in string, if pattern is found.

Comment : Characters in S and P denoted by subscripted s's and p's, respectively. The index variables used are i, the current guess at where P begins in S, and j and k, the indexes for S and P, respectively. The algorithm refers to the length n and m of the strings S and P, respectively. We assume that these are given as input and the end of the strings are marked by an end marker and that references to n and m can be replaced by tests for the end of the strings.

1. i $\leftarrow$ 0
2. while i $<n$ do
3. i $\leftarrow$ i+1 , j $\leftarrow$ i, k $\leftarrow$1
4. while $s_j = p_k$ do
5. if k = m then return SUCCESS

```
6.      else do j ← j+1 , k ← k+1 end
        end
    end
7. return FAILURE
```

If the length of each string is given, the test in line 2 could read "while $i \leq n-m+1$" to terminate the algorithm when i gets too close to the end of S for an occurrence of P to be possible.

It can be easily seen that m comparisons must be performed if the pattern appears at the beginning of the text. If P is not in S at all, n-m + 1 comparisons are done. There are cases in which the algorithm works quickly. What is the worst case? The numbers of comparisons would be maximized if for each value of i (that is, each possible starting place for P in S) all but the last character of P match the corresponding text characters. Thus the number of character comparisons in the worst case is at most m(n-m + 1) , which is not linear in the length of the strings involved.

2.3.2 Knuth-Morris Pratt Algorithm

A more efficient way to solve the classical pattern matching problem of finding the pattern in the text was suggested by Knuth, Morris and Pratt [9]. It is linear in the length of the strings involved.

The KMP-algorithm works in two phases. In the preprocessing phase, the algorithm constructs from the pattern P a so-called pattern-matching machine.

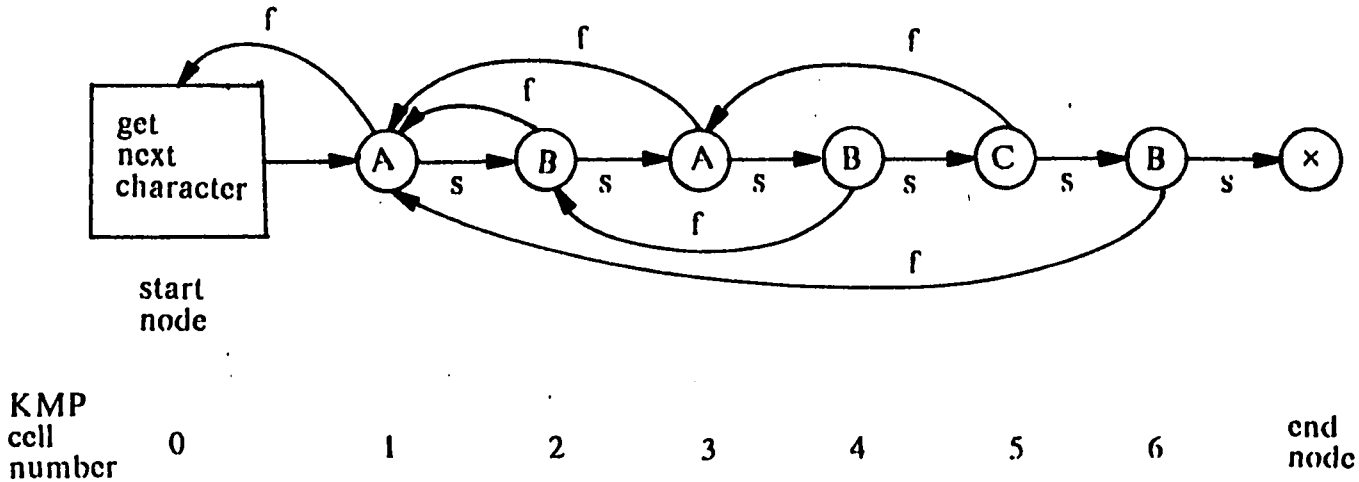


Figure 2.1 Flow diagram constructed by the KMP algorithm in the pre-processing phase.

Example 2.5 : Action of the KMP flow diagram in Fig. 2.1 for the pattern 'ABABCB' on the text 'ACABAABABA'

Table (2.1) Action of the KMP flow diagram :

KMP Cell number	Index of Text being scanned	Text Character being scanned	Success(s)/ Failure(f)
1	1	A	s
2	2	C	f
1	2	C	f
0	2	C	get next character
1	3	A	s
2	4	B	s
3	5	A	s
4	6	A	f
2	6	A	f
1	6	A	s
2	7	B	s
3	8	A	s
4	9	B	s
5	10	A	f
3	10	A	s
4	end of text		failure

Pre-processing time required is O(m). The pattern-matching machine is in fact, a finite-state automaton. The automaton has m + 1 states.

Fig. 2.1 gives the flow diagram of the pattern matching machine constructed by the KMP algorithm during the pre-processing phase for the pattern P = 'ABABCB'. The flow diagram contains two kinds of arrows - one of them is to be followed if the desired character is read from the text and the other arrow is to be followed if the desired character is not read from the subject. The arrows are called the success links and the failure links, respectively. The next character from the text is read only after a success link has been followed; the same text character is reconsidered if a failure link is followed; there is a starting node which causes a new text character to be read; and the starting point (first state) is the node that corresponds to the first pattern character. The last node marked as × is the final state and indicates that the pattern was found in the text.

In the second phase the KMP algorithm uses the flow diagram to scan the text S, spending time O(n). This is illustrated in Table 2.1. The scan terminates successfully if the last state, i.e., node marked × is reached, else if the end of the text is reached elsewhere in the text the scan terminates unsuccessfully.

2.3.2.1 Construction of the KMP Flow Diagram

Let FLINK be the array of failure links. FLINK(i) will be the index of the node pointed to by the failure link at the ith node, for $1 \leq i \leq m$. The special node which merely forces the next text character to be read is considered to be the

zero-th node, FLINK(1) = 0. To see how to set the other failure links consider an example with P='ABABCB'. Suppose the first four characters of P have matched 4 consecutive text character as indicated

Pattern P:ABABCB

Text S:ABABX...

Suppose that the next text character, X, is not a C. The next possible place where the pattern could begin in the text is at the third position shown, that is, as follows:

Pattern P: ABABCB

Text S:ABABX...

The pattern is moved forward so that the longest initial segment which matches part of the text preceding X is lined up with that part of the text. Now X should be tested to see if it is an A; in particular, the second A of the pattern. Thus the failure link for the node containing the C should point to the node containing the second A.

In general, the failure link of the ith node is set to the number j, which satisfies :

1. $j < i$
2. The first j-1 characters in the pattern match the j-1 characters that precede the ith node, i.e., $p_1, p_2, \ldots, p_{j-1}$ match $p_{i-(j-1)}, \ldots, p_{i-1}$.

3. j is the largest integer satisfying (1) and (2).

Note that j=1 obviously satisfies (2). Condition (3) provides for the maximum overlap of an initial segment of the pattern with the part of the text just scanned.

An algorithm for proper setting of the failure links can now be developed. This corresponds to the pre-processing phase. Suppose that the first i-1 failure links have been set. By condition (2) and with j = FLINK (i-1) the following segments of P must match :

$$\begin{array}{llll} p_{i-FLINK(i-1)} & p_{i-FLINK(i-1)+1} & \cdots & p_{i-2} \\ p_1 & p_2 & \cdots & p_{FLINK(i-1)-1} \end{array} \qquad (2.1)$$

If node i in the flow diagram is reached when scanning a text string, the last text character read had to be p_{i-1}. If $p_{i-1} = p_{FLINK(i-1)}$ the two matching sequences in expression 2.1 can be extended by one more character to get :

$$\begin{array}{llll} p_{i-FLINK(i-1)} & p_{i-FLINK(i-1)+1} & \cdots & p_{i-2} \quad p_{i-1} \\ p_1 & p_2 & \cdots & p_{FLINK(i-1)-1} \quad p_{FLINK(i-1)} \end{array} \qquad (2.2)$$

i.e., if $p_{i-1} = p_{FLINK(i-1)}$ then set FLINK(i) to FLINK(i-1)+1. In Fig. 2.1 FLINK(4) = 2 because p_1 matches p_3. Since $p_2 = p_4$, FLINK(5) = 3. If on the other hand $p_{i-1} \neq p_{FLINK(i-1)}$ as in Fig. 2.1 for i=6; we must find an initial substring of P that matches a substring ending at p_{i-1}. The match in expression

2.1 cannot be extended, so we look further back. Let j =FLINK(i-1) and $j' = FLINK(j)$. Then we have from the properties of the failure links :

$p_{(i-j)} \cdots p_{i-j} \cdots\cdots p_{i-2} \quad p_{i-1}$

$p_1 \quad \cdots p_{j-j'+1} \quad \cdots p_{j-1} \quad$ not p_{i-1}

$p_1 \quad \cdots p_{j'-1} \quad ?$

Clearly if $p_{j'} = p_{i-1}$ we would have an initial substring to match a substring ending at p_{i-1} and FLINK(i) should be $j'+1$. If $p_{j'} \neq p_{i-1}$ we must follow the failure link from node j' and try again. This process is continued until we find a failure link j such that $p_j = p_{i-1}$ or (as in Fig. 2.1 for i=6) j=0. In either case FLINK(i) should be j+1.

Algorithm : KMP "Flow Diagram" Construction

Input : P, a string of characters; $m \geq 1$, the length of P.

Output : FLINK, the array of failure links.

```
1.  FLINK(1) ←0, i←2
2.  while i≤m do
3.        j ← FLINK(i-1)
4.     while  j ≠  0 and p_j ≠ p_{i-1}
5.        do j← FLINK(j) end
6.        FLINK(i)← j+1
7.        i ← i+1
    end
```

2.3.2.2 The KMP Scan Algorithm

In the second phase of the KMP algorithm, the constructed flow diagram of the pattern P is used to scan the text S. This procedure of scanning has already been illustrated in example 2.5. The formal algorithm is as follows :

Algorithm : KMP Scan Algorithm

Input: P and S, the pattern and text strings; $m \geq 1$ and $n \geq 0$, their lengths; FLINK, the array of failure links set up in the flow diagram construction phase.

Output : A success or failure indicator.

1. i ← 1 , j ← 1 { j indexes flow diagram nodes; i indexes text characters }
2. while $i \leq n$ do
3. while $j \neq 0$ and $p_j \neq s_i$
4. do j ← FLINK(j) end
5. if j = m then return SUCCESS
6. else do i ← i+1 , j ← j+1 end

   end
7. return FAILURE

2.3.3 Modification of KMP Algorithm to Find Maximal Overlaps

Since the problem of finding the maximal pairwise overlaps between the strings in a given set of strings requires us to match the longest suffix of the text S which is also a prefix of the pattern P and vice versa, the KMP scan algorithm will always terminate in a FAILURE, assuming that the set is a substring free set. But after scanning the whole of S, the finite state machine is in state i if and only if $p_1 p_2 \ldots p_i$ is the longest suffix of S which is also a prefix of P. Hence i is the length of the maximal overlap between S & P. For instance, in example 2.5 (refer to Table 2.1) after the entire text-'ACABAABABA' is scanned the finite state machine is in state 3 (i.e., KMP cell number 3). Hence the maximum overlap between S = 'ACABAABABA' and P = 'ABABCB' is 3. Note that in state 4, there are no more text characters to be scanned and a failure is indicated.

In this way the maximal pairwise overlaps between any two strings s_i and s_j in a set of strings S $= \{s_1, s_2, \ldots s_n\}$ can be found in time O $(lg(s_i))$. By constructing the KMP flow diagram for each s_i and by scanning with all remaining strings all pairwise overlaps can be found.

2.3.4 Finite Input Memory Machine for Pattern Matching

In this section we investigate a method of finding overlaps by hardware.

We have seen that the Knuth-Morris-Pratt algorithm solves the classical

pattern matching problem of finding a pattern in a text string in linear time. Also we have seen that the KMP algorithm can be used to obtain the maximal overlaps in linear time. The question now is whether the KMP algorithm can be efficiently implemented in hardware. The answer is an emphatic no. To substantiate this answer let us look at the hardware which would be required to implement the KMP algorithm. The hardware required would be as follows: buffers to hold the pattern string and text string, hardware for computations of failure link settings in the preprocessing phase, registers to hold the failure link values i.e., FLINK(j) values (see sec 2.3.2.1), a single character comparator (k bit comparator for a character of k bits) for comparing a pattern character with a text character and control circuitry to dynamically skip through the text at each mismatch. Such a hardware scheme would become very complex. Schemes which are much simpler are possible in hardware. We present a scheme using a finite input memory machine.

The finite input memory machine implementation is quite straight forward and simple. It solves the classical pattern matching problem and also the overlapping problem in linear time. It does not require any preprocessing hardware and complex control circuitry as would be required in a KMP algorithm implementation. Also only a single output register is required; whereas in the KMP case, for a m character pattern, m registers are required to store the failure link data.

In Fig. 2.2 is shown a finite input memory machine. A finite input memory

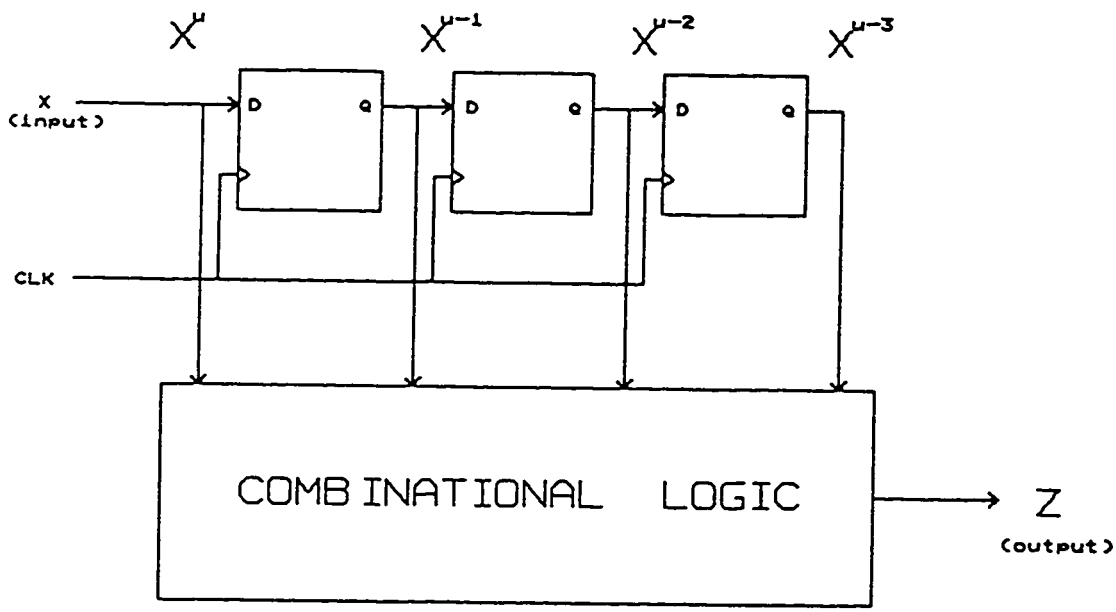


Figure 2.2 Finite input memory machine to identify a particular pattern.

machine or definite machine M of order u is a sequential machine, such that u is the least integer, so that the present output of M can be determined uniquely from the knowledge of the present and the last u inputs to M. The finite input memory machine in Fig. 2.2 is of order 3 since the past 3 input values are always sufficient to specify its present output. To solve the classical pattern-matching problem of finding a pattern in a text, we have to design the combinational logic part to identify a specific pattern. Let the pattern and text strings be over the binary alphabet, $\Sigma = \{0,1\}$. To match a pattern of length m we require a finite input memory machine of order m-1. Since our finite input memory machine in Fig. 2.2 is of order 3, our pattern can have a length m=4. Consider a 4 bit pattern P=1010. To find the occurrence of P=1010 in a binary text string S, the combinational logic part would have to be designed to identify P=1010. The truth table of the combinational logic part would then be as shown in Table 2.2. Note that the output is high only when $X^u X^{u-1} X^{u-2} X^{u-3} = 1010$, else it is low. Hence when the text string is applied to the input x, any occurrence of 1010 in the text stream would give a high output. If instead of binary strings, the strings consist of characters, then the characters would have to be encoded by bits. If k bits are required to encode each character, and if the pattern to be detected has m characters then k finite input memory machines each of order m-1 would be required.

Let us now turn our attention to finding the maximal overlaps between strings using a finite input memory machine. For the sake of simplicity,

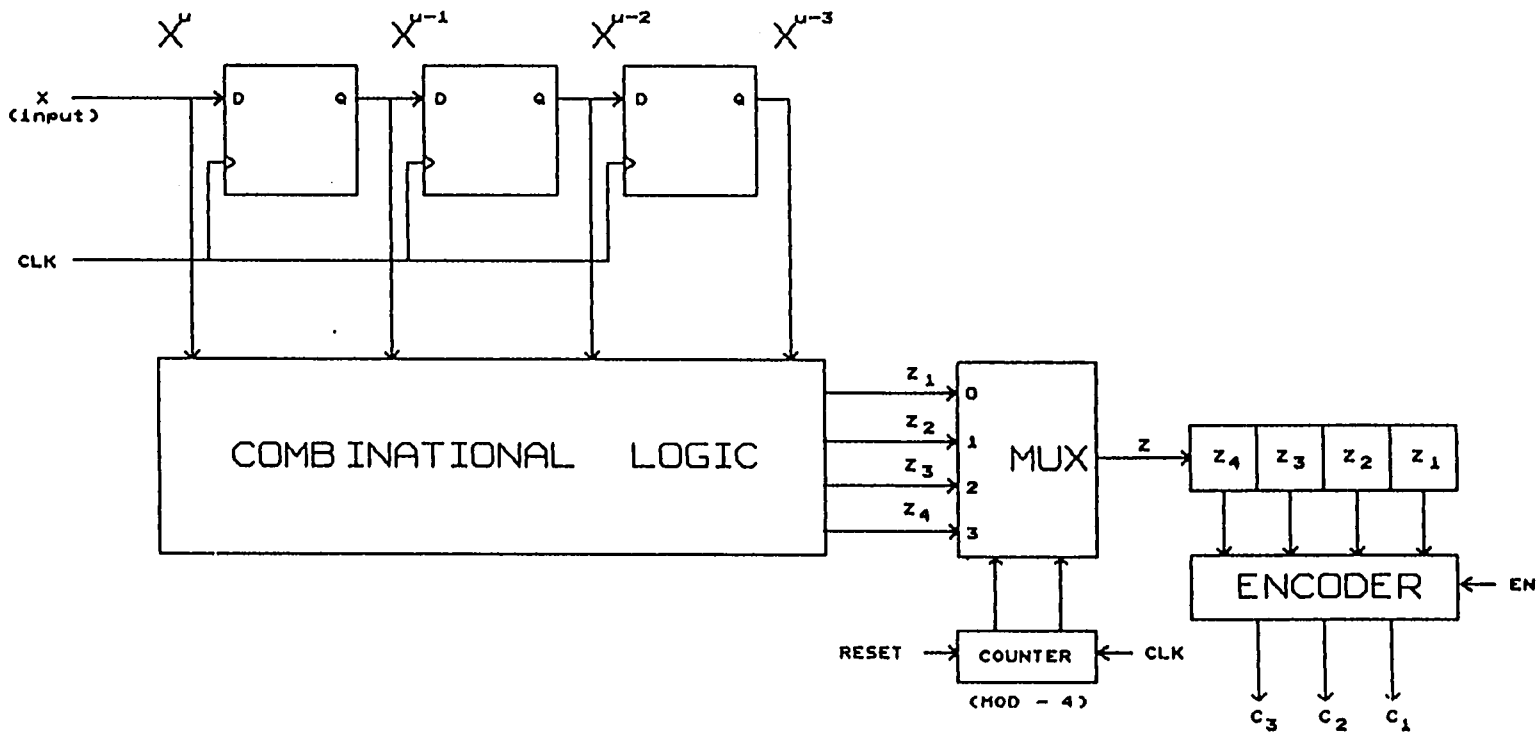
X^{n}	X^{n-1}	X^{n-2}	X^{n-3}	z
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Table 2.2 : Truth table for combinational logic of Fig. 2.2.

henceforth we will refer to the maximal overlap as just overlap. We will illustrate the procedure for finding overlaps by considering strings of length m=4 over the binary alphabet $\Sigma = \{0,1\}$. Consider $S = \{s_1,s_2,s_3,s_4\} = \{1000,1001,1011,1010\}$. We are interested in finding the overlap matrix for S. We will illustrate how to find the overlaps $\psi(s_4,s_1)$, $\psi(s_4,s_2)$ and $\psi(s_4,s_3)$. The scheme for finding these is shown in Fig. 2.3. Now the combinational logic part is designed to match the reversed pattern 0101, since finding $\psi(s_4,s_1)$, $\psi(s_4,s_2)$ and $\psi(s_4,s_3)$ involves matching the longest suffix of s_4 with the prefixes of s_1, s_2 and s_3 respectively. The truth table for the combinational logic part is shown in Table 2.3.

The counter in Fig. 2.3 is a mod-4 counter. The counter has an external reset, used to reset the counter at the time of starting.

To understand the working of the finite input memory circuit, let us consider that the input string arriving at the x-input is s_3 = 1011. The MSB of s_3 appears at the x-input first, followed by the other bits, down to LSB. Let the LSB bit of s_3 be numbered 0, and the next higher bit be numbered 1 and so on. So the MSB bit is numbered 3. At the start the counter is reset. Then the input and clock are simultaneously applied. When the present input X^u is bit 3 of s_3, the counter reads 00, and z_1 input of the multiplexer is selected. From the truth table (Table 2.3) we see that $z_1 = 0$ since bit 3 of s_3 equals 1. This value of z_1 is stored in an output shift register as indicated in Fig. 2.3. In the second clock



EN – Enable encoder every fourth clock pulse

Figure 2.3 Finite input memory machine to find overlaps.

X^w	X^{w-1}	X^{w-2}	X^{w-3}	z_1	z_2	z_3	z_4
0	x	x	x	1	x	x	x
0	1	x	x	x	1	x	x
0	1	0	x	x	x	1	x
0	1	0	1	x	x	x	1

Table 2.3 : Truth Table for Combinational logic of Fig. 2.3.

pulse the bit 2 of s_3 which is equal to 0 is available at the x-input. Since at this point $X^w = 0$ and $X^{w-1} = 1$, z_2 goes high and also z_1 goes high. But the counter reads 01 and the z_2 input of the multiplexer is selected; hence at this point it is immaterial whether z_1 is high or not. z_2 is stored in the output shift register and z_1 is pushed one place to the right. Next bit 1 of s_3 appears at x-input and from the truth table (Table 2.3) we see that $z_3 = 0$. The multiplexer selects z_3, since the counter reads 11 at this point. z_3 is shifted into the output shift register. Also z_2 and z_1 get shifted right one place. Similarly in the fourth clock pulse we have z_4 stored in the output shift register, shifting z_3, z_2 and z_1 one position to the right. Hence after four clock pulses z_4 occupies the MSB position in the output shift register and z_1 occupies the LSB position. The position of a '1' in the output shift register indicates the amount of overlap. For example, in the above case since $z_2 = 1$, the overlap is 2. The shift register contents can be encoded using a priority encoder after every four clock pulses to determine the overlap of the input stream with the string pattern for which the cell is designed. If say more than one value in the output shift register is 1 after the fourth clock pulse, then the higher bit position is taken as the maximal overlap. For instance if $z_3 = 1$ and $z_2 = 1$ simultaneously, then z_3 is taken and the overlap is 3. This is taken care of by the priority encoder whose truth table is given in Table 2.4. Note that if all of z_1 to z_4 are 0's then the overlap is 0. In general for a pattern of length m, we require m clock pulses to find an overlap. Since five possible overlaps are possible for the case m=4 we need three outputs for the priority

z_4 z_3 z_2 z_1	e_3	e_2	e_1
1 x x x	1	0	0
0 1 x x	0	1	1
0 0 1 x	0	1	0
0 0 0 1	0	0	1
0 0 0 0	0	0	0

Table 2.4 : Truth table of priority encoder.

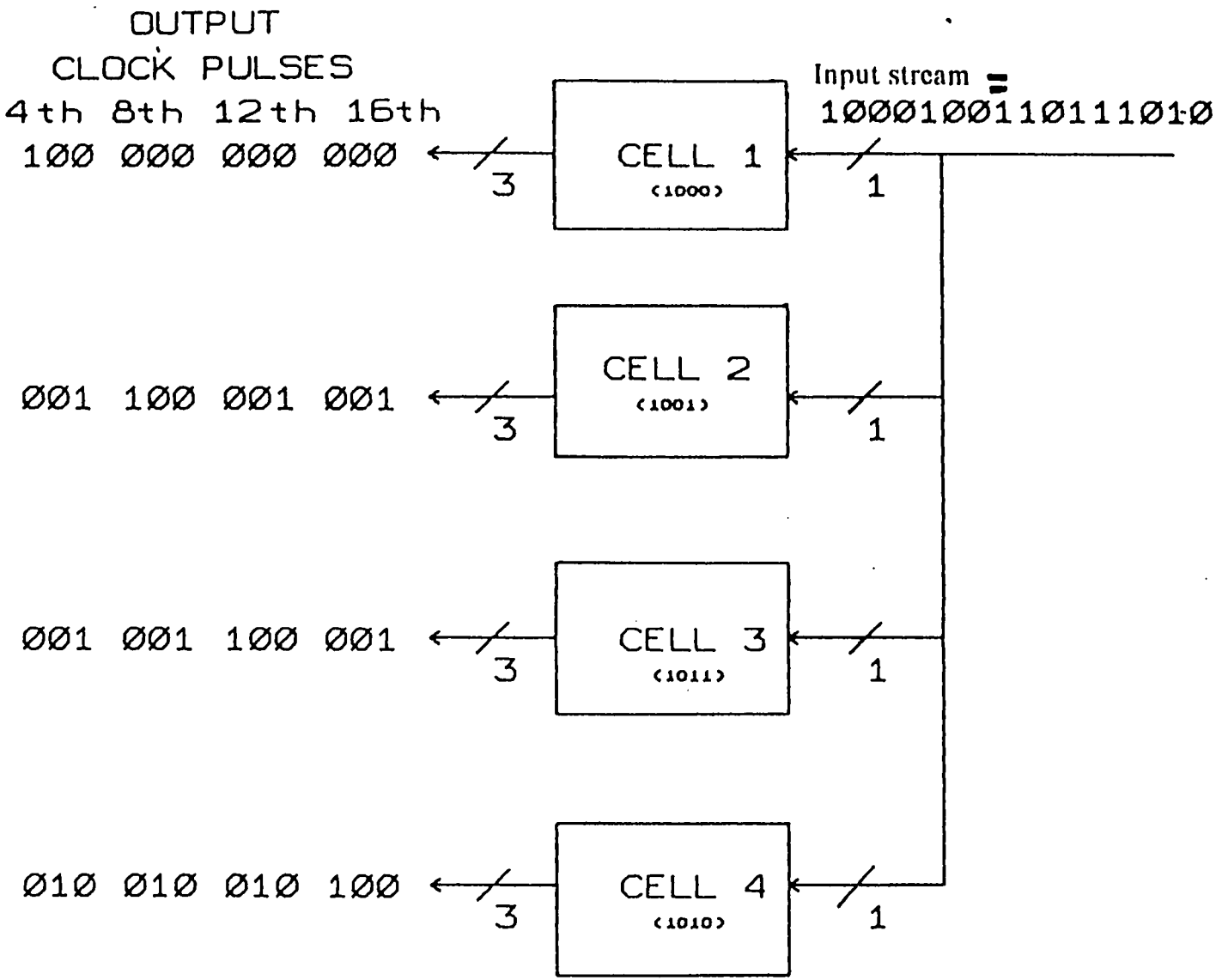


Figure 2.4 Basic cells in parallel.

encoder. Using similar basic cells as in Fig. 2.3, for each string in the set S, the overlap matrix for S = $\{s_1,s_2,s_3,s_4\}$ can be found. This is illustrated in Fig. 2.4. To extend the finite input memory machine design to find overlaps between character strings we would first require to encode the characters as bits. If there are k bits per character and if the pattern has m characters, then k finite input memory machines each of order m-1 would be required.

From the above discussion we can conclude that the KMP algorithm although good as a software implementation, its hardware implementation would be too complex. Hence alternative hardware implementations such as the finite input memory machine must be thought of. In chapter 4 we show that a systolic design to find the maximum overlaps is possible. This design is much more flexible than the finite input memory machine design and is well suited for VLSI implementation.

2.4 ALGORITHMS FOR SHORTEST COMMON SUPERSTRING

In this section we investigate several algorithms for the shortest common superstring problem. The superstring problem is NP-complete [5]. NP-complete is a class of problems, for which no polynomial bounded algorithm is known. For such problems, only an exhaustive case-by-case examination of all potential solutions will yield the optimal solution. But this would be too time-consuming and practically infeasible. We are therefore interested in polynomial-time approximation algorithms. Such algorithms construct a superstring that is not

necessarily a shortest one. All the algorithms investigated here produce solutions that are always within a factor of two of the optimum.

2.4.1 The Greedy Algorithm

Let us first understand the concept of a greedy algorithm [14,15]. Consider the problem of making change. Assume coins of values 25 cents, 10 cents , 5 cents, and 1 cent, and suppose we want to return 63 cents in change. Almost without thinking we convert this amount to two coins of 25 cents, one coin of 10 cents and three coins of 1 cent. Not only were we able to determine quickly the list of coins with the correct value, but we produced the shortest list of coins with the correct value.

The algorithm the reader probably used was to select the largest coin whose value was not greater than 63 cents, add it to the list and subtract its value from 63 getting 38 cents. Then the largest coin whose value was not greater than 38 cents (another 25 cents) was selected and added to the list and so on.

This method of making change is a **Greedy Algorithm.** At any individual stage a greedy algorithm selects that option which is "locally optimal" in some particular sense. Note that the greedy algorithm for making change produces an optimal global solution because of special properties of the coins. On the other hand, if the coins had values 1 cent, 5 cents, 11 cents and we were to make change of 15 cents, the greedy algorithm would first select an 11 cent coin and then four 1 cent coins for a total of 5 coins. However three 5 cent coins would

suffice and hence the greedy algorithm would not yield the optimal global solution.

Let us now consider the greedy algorithm to find the longest Hamiltonian path. First the overlap graph must be constructed as in examples 2.2 and 2.3. The greedy algorithm for finding the longest Hamiltonian path runs by finding one edge with maximal weight which together with the edges selected earlier can be expanded to a Hamiltonian path. When selecting an edge it is forbidden to select an edge that is not free any more. An edge (u,v) is called free if u is not the start node of some edge selected earlier and (u,v) together with the arcs selected earlier does not create an oriented cycle.

In more detail, let H be the set of arcs selected to yield a Hamiltonian path. Initially, H is empty. We proceed as follows.

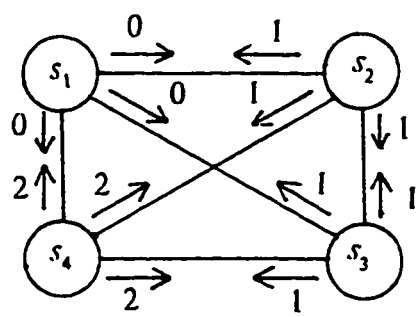
1. Sort the edges according to the weight .
2. Scan the edges in decreasing order. For each edge (u,v) encountered if (u,v) is free then:
   2.1 Add (u,v) to H, and
   2.2 Mark right(u) = v and left(v) = u (Initially right(u) = null and left(v) = null for all (u,v) belonging to the edgeset E), where right( ) and left( ) are two arrays. We further require two arrays rightend( ) and leftend( ). If left(u) = null then rightend(u) gives the vertex at the end of the path containing u in the current partial solution. Similarly if right(u) = null then leftend(u) gives the vertex at the

beginning of the path containing u in the current partial solution (Appendix 2.)

At the end of the scan, arrays left( ) and right( ) give the left and right neighbours of each vertex in the solution path.

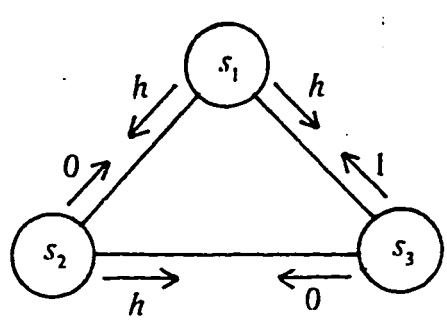
Consider the following two examples to illustrate the working of the greedy algorithm.

Example 2.6 : $s_1 = 1000$,$s_2 = 1001$,$s_3 = 1011$,$s_4 = 1010$



One possible solution of the greedy algorithm would be $\{s_4,s_3\}$, $\{s_3,s_2\}$ and $\{s_2,s_1\}$ in that order. When overlapping the strings in that order we get a superstring of length 12, yielding a reduction in length of 25%, compared to the case if the strings were just catenated without any overlapping.The optimal solution s_p also has length 12.

Example 2.7 : $s_1 = ab^h$, $s_2 = b^h b$,$s_3 = b^h a$



The greedy algorithm may select $\{s_1,s_3\}$ and $\{s_3,s_2\}$ producing a superstring ab^hab^hb of length $2h+3$, whereas the optimal solution is ab^hba of length $h+3$. Note, however, that the greedy method could have yielded the optimal solution if edge $\{s_1,s_2\}$ was selected first as it has the same weight as $\{s_1,s_3\}$. Thus the solution of the greedy algorithm depends on the selection of the initial edge.

An upper bound on the longest Hamiltonian path is given by the following Theorem. The proof of this theorem can be found in [4].

Theorem 2.1 : Let H be the approximate longest Hamiltonian path constructed by the greedy algorithm for the overlap graph of $S = \{s_1,s_2,\ldots,s_n\}$ and let H_{max} be a longest Hamiltonian path. Then $H \geq \frac{1}{2}H_{max}$. Therefore if s_g is the shortest common superstring constructed by the greedy method then $lg(s_g) \leq 2lg(s_p)$.

2.4.2 The Matching Algorithm

A matching in a graph G=(V,E) is a set of edges, no two of which share a common vertex. A maximum matching in a graph with edge weights wt(e) is a matching M that maximizes wt(M). A maximum matching M can be easily extended to a longest Hamiltonian path since the graph G is complete and a maximum matching must have total length at least half that of a longest path.

The matching algorithm to find SCS is defined as follows :

Algorithm : Match

Input : Directed complete overlap graph G(V,E), edge weights wt

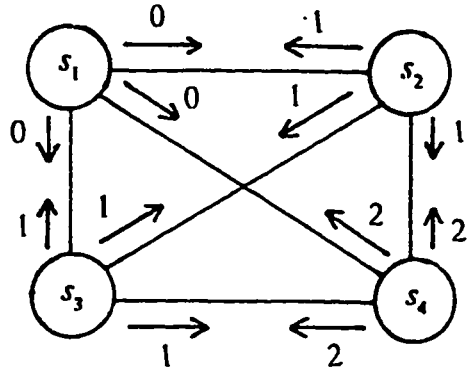
Output : Edgeset P

```
1. P := 0
2. while E ≠ empty
3.     do M := Maxmatch(G,wt)
4.        P := P U M;
5.        for (u,v) ∈ M
6.            Delete from G, all edges of the form (u,x) or (y,v)
7.            Collapse u and v into a single vertex
          end for
8. end while.
```

Let us illustrate the working of the matching algorithm by an example

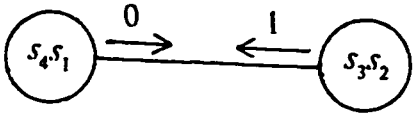
Example 2.8 : Let s_1, s_2, s_3 and s_4 be as in example 2.6 then from the overlap graph we can obtain an overlap table as shown below :

Step 1. Matching M : (s_4, s_1), (s_3, s_2)



	s_1	s_2	s_3	s_4
s_1	x	0	0	0
s_2	1	x	1	1
s_3	1	(1)	x	1
s_4	(2)	2	2	x

Step 2. Delete all entries of row s_4 and s_3 and columns s_1 and s_2. Collapse (s_4, s_1) to a single vertex and (s_3, s_2) to a single vertex.We get



	s_4s_1	s_3s_2
s_4s_1	x	0
s_3s_2	(1)	x

Step 3. Repeat step 1. New Matching M = (s_3s_2, s_4s_1)

Step 4. We obtain the solution P = $(s_3s_2s_4s_1)$

The superstring obtained is $s_p = s_3 \bullet s_2 \bullet s_4 \bullet s_1$ by overlapping the strings in the order s_3, s_2, s_4, s_1 and its length $lg(s_p)$ turns out to be 12. The same length as obtained by the greedy algorithm.

It has been shown in [7] that a minimum length superstring obtained by the matching algorithm is at most twice as long as that of an optimal length superstring. Finding a Maxmatch in a graph is very complicated and time-consuming. Hence we will not consider this algorithm any further.

2.4.3 Graph- Reduction Algorithm

This algorithm is intermediate between the greedy and the matching algorithm. The essential difference between this algorithm and the greedy

algorithm is that while in the greedy we operate on a single overlap graph to get a Hamiltonian path, here we reduce the graph successively as in the matching algorithm and at each step make a choice of overlap; i.e., select the maximum overlap.

The graph-reduction algorithm (Appendix 3) can be defined as follows :

Algorithm Graph-Reduction

Input : Directed complete overlap graph G(V,E), edgelengths l

Output : Edgeset P

1. P := 0
2. while E ≠ empty
3. do M := Max(G,l)
4. P := P U M;
5. for (u,v) ∈ M
6. Delete from G, all edges of the form (u,x) or (y,v)
7. Collapse u and v into a single vertex
   end for
8. end while.

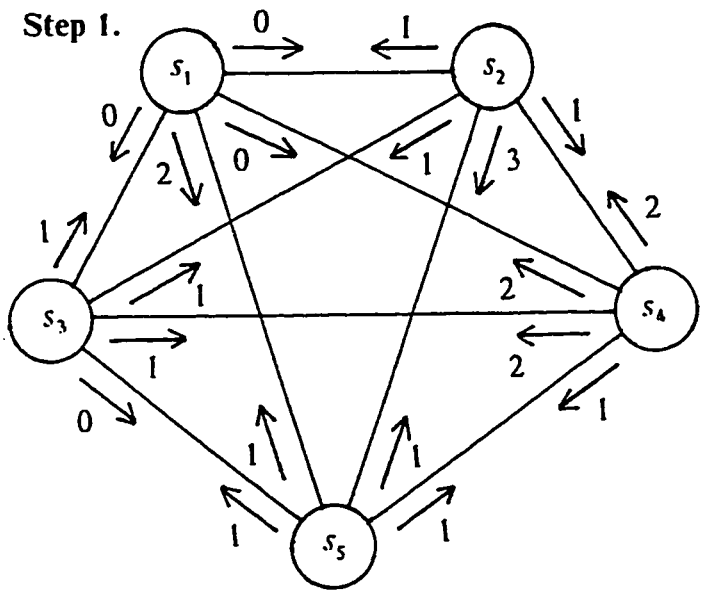
It can be seen that the essential difference between the matching algorithm and the graph-reduction algorithm is that in the former we find a max-matching; however in the latter algorithm we find only the maximum value. Also note that in the greedy algorithm we have to check for an oriented cycle,

this is not required here since step 7 of the algorithm leads to the formation of a newly-reduced graph each time.

Let us consider some examples to illustrate the working of this algorithm.

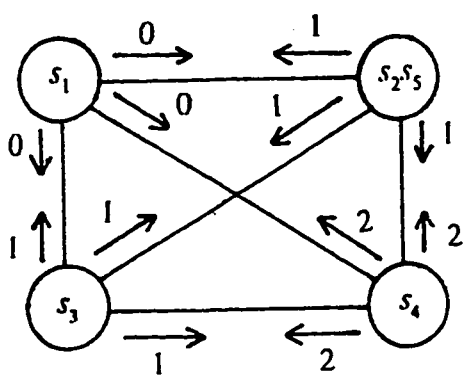
Example 2.9 : $S = \{s_1, s_2, s_3, s_4, s_5\} = \{1000, 1001\ 1011, 1010, 0011\}$

Step 1.



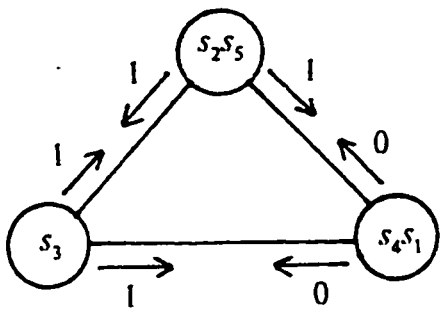
	s_1	s_2	s_3	s_4	s_5
s_1	x	0	0	0	2
s_2	1	x	1	1	③
s_3	1	1	x	1	0
s_4	2	2	2	x	1
s_5	1	1	1	1	x

Step 2.



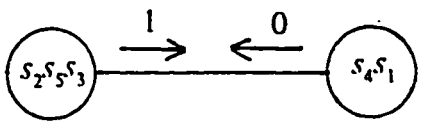
	s_1	s_2s_5	s_3	s_4
s_1	x	0	0	0
s_2s_5	1	x	1	1
s_3	1	1	x	1
s_4	②	2	2	x

Step 3.



	s_2s_5	s_3	s_4s_1
s_2s_5	x	(1)	1
s_3	1	x	1
s_4s_1	0	0	x

Step 4.

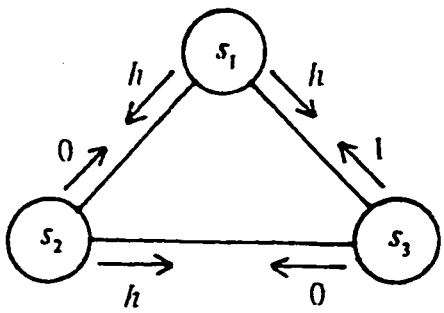


	$s_2s_5s_3$	s_4s_1
$s_2s_5s_3$	x	(1)
s_4s_1	0	x

The superstring is got by overlapping the strings in the order s_2, s_5, s_3, s_4 and s_1 and is 1001101101000. The length of the superstring is 13; yielding a reduction in length of 35%, compared to the case if the strings were just catenated without any overlapping.

An additional improvement incorporated in the graph-reduction algorithm that leads to better results in certain cases is illustrated by reconsidering example 2.7.

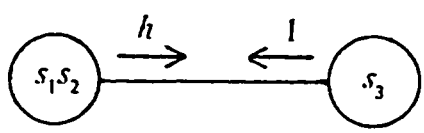
Step 1:



	s_1	s_2	s_3
s_1	x	$\textcircled{h}$	h
s_2	0	x	h
s_3	1	0	x

The improvement incorporated is that we try to avoid selections which would eliminate good selections later. For example in the overlap matrix above we could have selected the h in the first row and third column. But this would mean that no other h in the first row or third column can be selected. However by selecting the h in the second row and column, we leave open the possibility of selecting a further h as illustrated in step 2. In general we avoid selections which have all max values in the corresponding row and column at any step of the reduction process.

Step 2:



	s_1s_2	s_3
s_1s_2	x	$\textcircled{h}$
s_3	1	x

Step 3:

The superstring obtained is straightaway ab^hba, unlike in the greedy case.

The worst case bound of the superstring by the graph-reduction algorithm is the same as that of the greedy and matching algorithm, i.e., twice that of the optimum solution, since it is also based on selecting the maximum value at each step.

2.5 SUMMARY

The problem of finding a SCS for a given set of strings has been dealt with in detail. Both software and hardware methods to find overlaps have been discussed. However, the hardware implementation is not very efficient, being pattern-dependent; and also the software implementation is likely to be slow. Hence a more efficient hardware implementation based on a systolic array design is proposed later in Chapter 4. The graph-reduction algorithm for SCS has been proposed. In the next chapter the SCS problem is applied to a set of essential sequences [8], which yields shorter length checking experiments compared to conventional methods.

CHAPTER 3

DIGITAL TESTING AND CHECKING EXPERIMENTS

3.1 INTRODUCTION

Efficient techniques for testing and diagnosis of digital systems and modules are becoming of critical importance. Current trends for increased density of devices on integrated circuits are creating a situation in which system hardware costs are being significantly decreased due to this increased density. But testing is becoming more difficult since the number of pins per I.C. is not increasing in proportion to the number of gates per IC, thus reducing the observability and controllability of the logic on the chip.

Several methods exist for generating efficient tests for detecting faults in combinational circuits, but the generation of reasonable test sequences for detecting faults in sequential machines is not as straight forward as its combinational counterpart. The difficulty arises from the existence of different states in a sequential machine, which cause it to respond differently under the same inputs, and therefore adding another dimension of uncertainty.

Two different approaches to the problem of deriving test sequences for detecting faults in sequential machines have been investigated in the literature. In the first, referred to as the circuit - testing approach based in the classical methods of combinational testing, the structure of the machine and the class of faults that can occur are assumed to be known. Significant contributions based

on this approach have come from Poage and McCluskey [16], Seshu and Freeman [17] and Roth [18]. In the second approach, the transition table of the machine realization is known. The second approach is a functional testing approach, which aims at generating tests from test-object descriptions which are at higher levels than the gate. Its purpose is to detect the presence of a fault, but not the exact nature or location of the failure. A functional test should check that intended functions are executed and also that no unintended functions are performed.

An important parameter in testing is the time taken by the test. In order to accomplish testing speedily the length of the test sequence should be kept as small as possible. Although the circuit testing approach generally leads to shorter testing sequences than functional testing, it is much more tedious and has a lesser fault coverage.

In this chapter we are concerned with the second approach, i.e., the functional testing approach. The earliest work on the lines of functional testing was done by Moore [19]. His approach to the problem was to derive a sequence of inputs which distinguishes a normal machine represented by a transition table M_o from all other transition tables with the same number of states, inputs and outputs. The next contribution come from Hennie [20] who developed a simple and effective algorithm, to derive test sequences for sequential machines. Other significant contributions based on this approach have come from Hsieh [21], Friedman and Menon [22], Boute [23], Braun [24], Kohavi [1], Lin and Menon [25] and others. However the length of the test sequences obtained have often

been excessive even for a moderate number of states. Recently it has been shown in [8] that it is possible to drastically reduce the length of the test sequences by combining the state identification phase with the state transition verification phase and by judiciously overlapping of sequences verifying individual state transitions.

In this chapter, we will primarily look into the method of deriving test sequences as suggested in [8]. Specifically we will apply the algorithms developed for shortest common superstring in chapter 2 to construct efficient test sequences of possibly minimal length for a given sequential machine. We begin the chapter by some preliminaries on finite state machines and state identification sequences. This is followed by methods for deriving D-sequences. Finally checking experiments are developed by applying the graph-reduction algorithm to a set of essential sequences, of a sequential machine. The checking experiments thus obtained are compared with those obtained by Hennie's procedure.

3.2 FINITE STATE MACHINES

Definition

A *finite-state* or *sequential machine* is quintuple $M = (Q, A, Y, \delta, \lambda)$ where Q, A and Y are finite, non-empty sets (of cardinality in general greater than 1) of state, input and output symbols(or letters), respectively, and,

$$\delta : Q \times A \rightarrow Q$$

$$\lambda : Q \times A \rightarrow Y$$

are functions, the direct transition function and the direct output function,

respectively. These two functions can be combined into

$$\nu : Q \times A \rightarrow Q \times Y,$$

where $\nu(q,a) = (\delta(q,a),\lambda(q,a))$ for $q \in Q$ and $a \in A$.

The two functions of a machine may be extended to

$$\delta : Q \times A^* \rightarrow Q$$

$$\lambda : Q \times A^* \rightarrow Y^*$$

by the following recurrence relationships :

$$\delta(q,e) = q, \quad \delta(q,wa) = \delta(\delta(q,w),a) \tag{3.1}$$

$$\lambda(q,e) = e, \quad \lambda(q,wa) = \lambda(q,w)\lambda(\delta(q,w),a) \tag{3.2}$$

where $q \in Q$, $w \in A^*$, $a \in A$. The infinite sets A^* and Y^* are the sets of all finite-length sequences (or words, or strings) of symbols from A and Y, respectively, including the empty sequence e (of length 0). In a similar way the combined function can be extended to

$$\nu : Q \times A^* \rightarrow Q \times Y^*.$$

Note that the direct functions are special cases of the extended ones, so that the same notation for both is justified.

We can define three equivalences δ_w, λ_w, and ν_w by the following relations :

For $p,q \in Q$, $w \in A^*$

$$p \delta_w q \ <=> \ \delta(p,w) = \delta(q,w)$$

$$p \lambda_w q \ <=> \ \lambda(p,w) = \lambda(q,w)$$

$$p \nu_w q \ <=> \ \nu(p,w) = \nu(q,w)$$

We can also define a function

$\hat{\delta} : P(Q) \times A \rightarrow P(Q)$ by

$p\hat{\delta}_w(\pi)q \iff \delta(p,w)\pi\delta(q,w)$, where $\pi \in P(Q)$, and $P(Q)$ is the set of all equivalence relations or partitions on the state set Q.

The following properties hold directly.

1. $\hat{\delta}_w(\varphi) = \delta_w$ (3.3)
2. $\hat{\delta}_w(\omega) = \omega$ (3.4)
3. $\pi \leq \sigma = \hat{\delta}_w(\pi) \leq \hat{\delta}_w(\sigma)$ (3.5)
4. $\hat{\delta}_w(\pi \wedge \sigma) = \hat{\delta}_w(\pi) \wedge \hat{\delta}_w(\sigma)$ (3.6)
5. $\nu_w = \lambda_w \wedge \delta_w$ (3.7)

where φ and ω are the identity and universal relation on Q respectively, $\pi, \sigma \in P(Q)$, $w \in A^*$.

The three equivalence relations can be determined by the following recurrence relationships :

$$\delta_e = \varphi \quad \delta_{aw} = \hat{\delta}_a(\delta_w) \tag{3.8}$$

$$\lambda_e = \omega \quad \lambda_{aw} = \lambda_a \wedge \hat{\delta}_a(\lambda_w) \tag{3.9}$$

$$\nu_e = e \quad \nu_{aw} = \lambda_a \wedge \hat{\delta}_a(\nu_w) \tag{3.10}$$

where $a \in A$, and $w \in A^*$.

Definition (Reduced Machines)

A machine $M = (Q, A, Y, \delta, \lambda)$ is said to be reduced iff

$(\forall p,q \in Q), \exists w \in A^*, \ \lambda(q,w) \neq \lambda(p,w)$

Otherwise, M is not reduced, and p, q are equivalent states.

Definition (Strongly Connected Machines)

A machine $M = (Q, A, Y, \delta, \lambda)$ is said to be strongly connected iff

$$(\forall p,q \in Q), \exists w \in A^*, \ \delta(p,w) = q$$

In other words, the machine M can be taken from any state p in Q to any other state q in Q.

3.3 STATE IDENTIFICATION SEQUENCES

State identification sequences are input sequences to a sequential machine M which produce output sequences that allow the identification of either the unknown initial state of M or the final state of M.

Let $M = (Q, A, Y, \delta, \lambda)$ be a sequential machine.

Definition (Synchronizing Sequence or S-Sequence)

An input sequence $w \in A^*$ is said to be a synchronizing sequence for M if it takes M to a particular final state q_w regardless of the initial state of M ; formally, if for $q_w \in Q$

$$(\forall q \in Q), \delta(q,w) = q_w \tag{3.11}$$

Since w will drive all the states to one single state, then

$$\delta_w = \omega \tag{3.12}$$

where ω is the universal relation on Q.

Definition (Homing Sequence or H-Sequence)

An input sequence $w \in A^*$ is said to be a homing sequence for M if the output sequence of M in response to w is different, whenever the final state of M is different, regardless of the initial state of M ; formally, if

$$(\forall\, p,q \in Q),\ \delta(p,w) \neq \delta(q,w) \ \Rightarrow\ \lambda(p,w) \neq \lambda(q,w) \tag{3.13}$$

It has been shown in [24] that w is a homing sequence if

$$\nu_w = \lambda_w \tag{3.14}$$

Definition (Distinguishing Sequence or D-Sequence)

An input sequence $w \in A^*$ is said to be a distinguishing sequence for M if the output sequence of M in response to w is different for every different initial state of M ; formally, if

$$(\forall p,q \in Q),\ p \neq q \ \Rightarrow\ \lambda(p,w) \neq \lambda(q,w) \tag{3.15}$$

Since each state yields a different output under w

$$\lambda_w = \varphi \tag{3.16}$$

i.e., the function $\lambda_w : Q \rightarrow Y^*$ is injective. Note also that every distinguishing sequence is also a homing sequence.

Definition (Distinguishing Prefix or D-Prefix)

An input sequence $w \in A^*$ is a distinguishing prefix for M if the pair of output sequence and final state of M in response to w is different for every different initial state of M ; formally if

$$(\forall p,q \in Q),\ p \neq q \ \Rightarrow\ \nu(p,w) \neq \nu(q,w) \tag{3.17}$$

If w is a D-prefix, then

$$\nu_w = \varphi \tag{3.18}$$

i.e., the function $\nu_w : Q \rightarrow Q \times Y^*$ is injective.

Every D-sequence is a D-prefix since from equation (3.7) we have

$$\lambda_w = \varphi \Rightarrow \nu_w = \varphi$$

but the converse does not hold.

In the derivation of testing sequences for sequential machines we will mainly be dealing with D-sequences. The S-sequences and H-sequences are required only for initialization purpose; i.e., to drive the machine to a known initial state. Therefore we will consider here only the D-sequences in detail.

3.4 D-SEQUENCES

3.4.1 Derivation of D-sequences

There are basically two methods of deriving distinguishing sequences. The first method based on the development of a successor tree has been discussed extensively in the literature [1,27], therefore we shall not study it here in detail. An illustrative example taken from [1] is given in the next page for this method.

Example 3.1 :

Consider

M_1	0	1
A	B,0	C,1
B	C,0	D,0
C	D,1	C,1
D	A,1	B,0

Fig. 3.1 is the successor tree for machine M_1. It has 3 levels as shown clearly in Fig.3.1 It is composed of nodes and branches. The nodes are labelled (ABCD), (BC)(AD), etc.. and the branches are the lines joining the nodes between any two successive levels. Each branch is labelled with the input symbol it represents. M_1 can initially be in any of its 4 states. In such a case we say that the initial uncertainty regarding the state of M_1 is given by the set of states (ABCD). Successive inputs reduce this initial uncertainty, leading to lesser uncertainties. Generally a collection of uncertainties is called an uncertainty vector. For example in Fig. 3.1, (BC)(AD) is an uncertainty vector. The individual uncertainties contained in the vector are called the components of the vector. For example (BC) and (AD) are components of (BC)(AD). An uncertainty vector whose components contain a singic state each is said to be a trivial uncertainty vector. For example (C)(D)(A)(B) is a trivial uncertainty vector. If a component of an uncertainty vector has identical repeated states then it is called a homogeneous component. For example the uncertainty vector

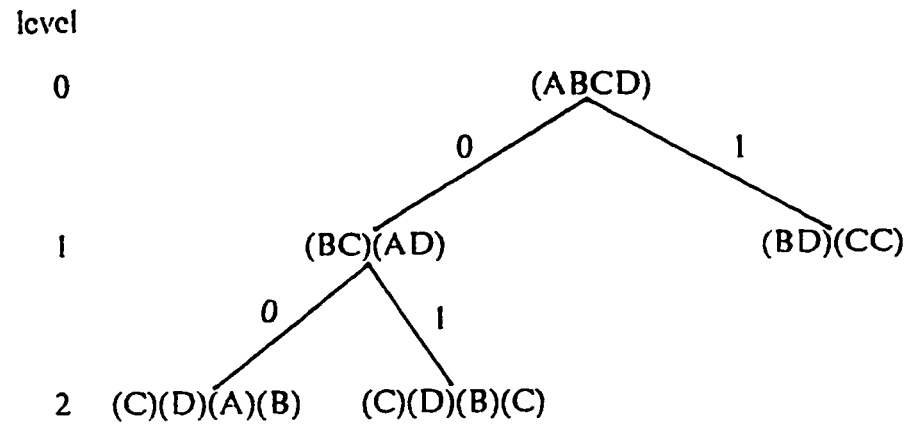


Figure 3.1 Successor tree to find D-sequence.

(BD)(CC) has a non-trivial homogeneous component since the component (CC) has repeated states.

In order to find a distinguishing sequence from a successor tree, it has to be properly pruned. The conditions under which a node in any level except 0 level of the successor tree becomes terminal are :

1. The node is associated with an uncertainty vector whose non-homogeneous components are associated with some node in a preceding level.

2. The node is associated with an uncertainty vector containing a homogeneous nontrivial component.

3. The node is associated with a trivial uncertainty vector.

The machine will possess a D-sequence if the pruned successor tree has a trivial uncertainty vector associated with it at some node. Then the D-sequence for the machine is obtained by starting from the root node, and traversing the branches up to the node containing the trivial uncertainty vector. Specifically the catenation of the input symbols on the branches traversed from the root node to the node containing the trivial uncertainty vector gives the D-sequence. For M_1, from Fig. 3.1 we can deduce that the D-sequences are 00 and 01. Note that a machine may have a number of D-sequences of differing lengths (no. of input symbols). In the derivation of test sequences we are interested only in minimal length D-sequences.

As part of the thesis work, a recursive program has been developed to find a minimal length D-sequence by the above discussed method and is listed in Appendix 4.

The second method uses equation (3.16), and the recurrence relation (3.8), to develop a so-called D-tree. A node u in the D-tree is terminated if one of the following two conditions is satisfied :

1. $\exists w \in A^{*}, \quad \lambda_w = \lambda_u \quad |w| \leq |u|$ (3.19)

2. $\lambda_u = \varphi$ (3.20)

If condition (1) is satisfied, that node need not be developed any further, since it already appeared previously.

If, on the other hand, condition (2) is satisfied, then the D-sequence can be found by reading the inputs from the leafs up towards the root. The inputs have to be read in this way because the recurrence relation (3.8) pre-catenates a new input to w at each step of the formation of the D-tree.

Example 3.2 :

Consider

M_1	0	1
A	B,0	C,1
B	C,0	D,0
C	D,1	C,1
D	A,1	B,0

0	1
D	
A	D
B	A,C
C	B

Note that the second half of the state table indicates the previous states, which will be used in obtaining $\hat{\delta}(\pi)$. The D-tree (fig 3.2) can be developed using the relations (3.9).

$\lambda_0 = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$

$\lambda_1 = \overline{B}\,\overline{D}\,\overline{A}\,\overline{C}$

$\lambda_{00} = \lambda_0 \Lambda \hat{\delta}_0(\lambda_0) = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} \Lambda \overline{A}\,\overline{D}\,\overline{B}\,\overline{C} = \overline{\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}} = \varphi$

$\lambda_{01} = \lambda_0 \Lambda \hat{\delta}_0(\lambda_1) = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} \Lambda \overline{A}\,\overline{C}\,\overline{B}\,\overline{D} = \overline{\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}} = \varphi$

$\lambda_{10} = \lambda_1 \Lambda \hat{\delta}_1(\lambda_0) = \overline{B}\,\overline{D}\,\overline{A}\,\overline{C} \Lambda \overline{D}\,\overline{A}\,\overline{B}\,\overline{C} = \overline{D}\,\overline{B}\,\overline{A}\,\overline{C}$

$\lambda_{11} = \lambda_1 \Lambda \hat{\delta}_1(\lambda_1) = \overline{B}\,\overline{D}\,\overline{A}\,\overline{C} \Lambda \overline{B}\,\overline{D}\,\overline{A}\,\overline{C} = \overline{B}\,\overline{D}\,\overline{A}\,\overline{C}$

From the tree, 00 and 01 are the shortest D-sequences for M_1. A program for finding the D-sequences of a machine by the second method is listed in Appendix 5.

3.4.2 Existence of D-sequences

It has been shown in [26], that a necessary condition for the existence of a distinguishing sequence is

$$\Xi a \in A \quad v_a = \lambda_a \Lambda \delta_a = \varphi \tag{3.21}$$

and, a sufficient condition for the existence of a distinguishing sequence is that

$$\forall a \in A \quad v_a = \varphi \tag{3.22}$$

The interested reader is referred to [26] for proofs of the above.

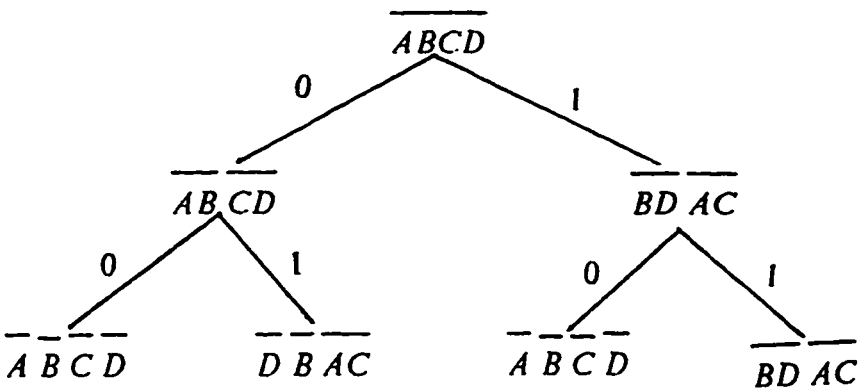


Figure 3.2 D-tree to find D-sequence.

3.5 COMPARISON OF METHODS FOR DERIVING D-SEQUENCES

We have discussed two methods of deriving D-sequences . The first method used sets and successor states, and the other used partitions and predecessor states . While the first method requires us to check for three conditions (as outlined in section 3.4) at each node to determine, whether a node is terminal or not, the second method requires us to check for only two conditions (equations 3.19 and 3.20) at each node. The second method is more efficient when the same number of nodes have to be checked to determine the D-sequence.

Additionally in the second method an initial check by inspection (equation 3.22) will tell us whether a D-sequence may exist for the machine or not. This check has been incorporated in the program in appendix 5 .

A third method proposed by Das et al [28] uses transition matrices and submatrices to derive the D-sequences. Their method does not guarantee a minimal length sequence and for some cases, is unable to find the complete set of D-sequences, also the complexity grows exponentially with the number of states as well as the length of the experiment.

3.6 CHECKING EXPERIMENTS

Definition (Checking Experiments)

A checking experiment is an input-output sequence which distinguishes a machine M from all other machines with the same number of states, inputs and outputs. In other words, a checking experiment determines whether the state table of the machine M is realized or not.

Checking experiments can be adaptive or preset depending upon whether or not the next input to be applied is based on the outputs previously produced by the machine.

Hennie [20] used the state identification sequences to arrive at a general checking experiment that would detect any fault in a machine M that does not increase the number of states of M and satisfies the following conditions :

1. M is strongly connected.
2. M is state reduced.
3. M has D-sequences.

He divided the experiment into two parts : initialization and diagnosis.

Initialization is done using an S-sequence, if it exists. In such a case the experiment will be totally preset (both initialization and diagnosis). Since all machines do not have an S-sequence, it is sometimes necessary to use a H-sequence. It has been shown in the literature [1] that a H-sequence, whose length is at most $(n-1)^2$, exists for every reduced n-state machine M. Since M is assumed to be state reduced, it is guaranteed to possess a H-sequence [1]. The diagnosis part of the experiment is often designed with a particular initial state (say q_o) in mind. Hence during initialization if the H-sequence drives the machine to a state other than q_o, a transfer sequence is required, to force the machine to state q_o. Formally we define :

Definition (Transfer Sequence)

Let $M = (Q,A,Y,\delta,\lambda)$ be a sequential machine. A transfer sequence T(p,q),

where $p,q \in Q$, is the shortest input sequence that takes M from state p to state q.

In the diagnosis part, the number of states of the machine are verified and each transition is tested to be consistent with the state table. Verification of states can be done by applying a D-sequence at each state and concluding from the different responses that the different states exist.

The general procedure by Hennie is discussed in [1.29], therefore we shall not deal with it in detail. From now on we assume that all the machines are strongly connected, state reduced and possess a D-sequence. Consider an example to illustrate Hennie's procedure.

Example 3.3 :

Consider M_1 given in example 3.1. The checking experiment obtained using Hennie's procedure and using the D-sequence 01 is shown below.

The final checking experiment is

$A^0B^1D^0A^1C^0D^1B^0\ C^1C^0D^1B^0C^0D^1B^1\ D^0A^1C^0D^0A^1C^0D^0\ A^0B^1D^0A^0B^0C^1C$

or in more detail :

input : 0 1 0 1 0 1 0 1 0 1 0 0 1 1 0 1 0 0 1 0 0

output : 0 0 1 1 1 0 0 1 1 0 0 1 0 0 1 1 1 1 1 1 1

time : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

input : 0 1 0 0 0 1

output : 0 0 1 0 0 1

time : 21 22 23 24 25 26 27

The first eight output symbols, by displaying four different responses to the D-sequences 01, i.e. 00, 11, 10 and 01 verify that M has indeed four distinct states. The next two input symbols, i.e., the ninth and the tenth guarantee that the machine terminates in state B, since it has already been established that a response of 10 to the distinguishing sequence indicates a transition from state C to state B. From this point on, if at any time during the course of the experiment one of the above responses to the D-sequences is produced, the state of the machine at that time is uniquely identifiable. Notice that at time = 10, 13, 16, 20 and 24 five transitions are verified explicitly and at time = 5, 7, and 15 the remaining transitions are implicitly verified. Notice that at time = 19 and 23 we require transfer sequences T (C,D) and T(D,A). The length of this experiment is 27. Initialization is achieved by a H-sequence 00, followed by either one of the transfer sequences T(C,A) = 00, T(D,A) = 0 and T(B,A) = 10. In case the H-sequence leaves the machine in state A, there is no need for a transfer sequence. If the H-sequence leaves M in a state different from A then a transfer sequence is required. In this case the initialization part has a length 4 in the worst case and the total length of the experiment becomes 31. This length however is much longer than the optimal length, and shorter checking experiments can be obtained by other methods.

3.7 SUPERSEQUENCE CONSTRUCTION METHOD

In [8] a method of deriving checking experiments has been suggested which uses what are called essential sequences. We propose here to use the essential sequences as defined in [8] and use the Graph-Reduction Algorithm (discussed in Chapter 2) to obtain an optimal or near-optimal checking experiment.

Definition (State-Input Sequence or SI Sequence)

Let $M=(Q,A,Y,\delta,\lambda)$ be a sequential machine. The function $\tilde{\delta}: Q\times A^{*}\rightarrow (Q\times A)^{*}\times Q$ is defined recursively as follows

$$\tilde{\delta}(q,e)=q \tag{3.23}$$

$$\tilde{\delta}(q,aw)=(q,a)\tilde{\delta}(\delta(q,a),w) \tag{3.24}$$

where $q\in Q$, $a\in A$ and $w\in A^{*}$. In particular,

$$\tilde{\delta}(q,a)=\tilde{\delta}(q,ae)=(q,a)\tilde{\delta}(\delta(q,a),e)=(q,a)\delta(q,a)$$

The sequence $\tilde{\delta}(q,w)$ is a sequence of alternating state and input symbols, starting with state q and terminating in a state $\tilde{\delta}(q,w)$, and is called a *state-input sequence* or SI-sequence for short.

Definition (Essential Sequences)

Let $M=(Q,A,Y,\delta,\lambda)$ be a sequential machine and let $d\in A^{*}$ be a (minimal-length) D-sequence of M. Then we may associate to M a so-called d-machine $M_d=(Q,A_d,Y_d,\delta_d,\lambda_d)$, where $A_d=A.d$, $Y_d=\{\lambda(q,ad)|q\in Q,a\in A\}$, and δ_d, λ_d are the sets A_d^{*}, Y_d^{*} restricted transition and output function, respectively. Furthermore, the set of SI sequences $ES_d=\{\tilde{\delta}(q,w)|w\in A_d\}$ is called the set of essential sequences. The function $\tilde{\delta}$ generates SI sequences.

At times we need to filter out the "underlying" input sequence. This reverse operation is accomplished by the function $s:(Q\times A)^{*}\times Q\rightarrow A^{*}$, defined as follows

$$s(\tilde{\delta}(q,w))=w \tag{3.25}$$

Example 3.4 :

Consider M_1 using the D-sequence d=01 We can write the essential sequences as follows :

$$\tilde{\delta}(A,0d) = A_0 = A^0B^0C^1C$$

$$\tilde{\delta}(B,0d) = B_0 = B^0C^0D^1B$$

$$\tilde{\delta}(C,0d) = C_0 = C^0D^0A^1C$$

$$\tilde{\delta}(D,0d) = D_0 = D^0A^0B^1D$$

$$\tilde{\delta}(A,1d) = A_1 = A^1C^0D^1B$$

$$\tilde{\delta}(B,1d) = B_1 = B^1D^0A^1C$$

$$\tilde{\delta}(C,1d) = C_1 = C^1C^0D^1B$$

$$\tilde{\delta}(D,1d) = D_1 = D^1B^0C^1C$$

The first step in applying the shortest common superstring method is to find the matrix of overlaps (as explained in chapter 2) of essential sequences. Note that the essential sequences are SI sequences, the basic elements of which are now no longer individual letters $a \in A$ but pairs $(q,a) \in Q \times A$, except for the last element which is always a state without the accompanying input letter. It is assumed that a "wild card" or "don't care letter", is associated with the final state, which can be replaced by any letter in A (the input set). However, this don't care input is not indicated explicitly, its presence being indicated by an empty space. For the sake of convenience we will not take into account the final state and the don't care input associated with it when determining the maximal overlaps. Then the resulting overlap corresponds to the overlap of the

underlying input sequences. Hence the overlap matrix for the essential sequences of M_1 is as shown :

	A_0	B_0	C_0	D_0	A_1	B_1	C_1	D_1
A_0	x	-2	0	-1	-2	-2	1	-1
B_0	-2	x	-1	-1	-2	0	-1	1
C_0	-2	-2	x	-1	1	-2	0	-1
D_0	-1	-1	-2	x	-1	1	-2	0
A_1	-2	0	-1	-1	x	0	-1	1
B_1	-2	-2	0	-1	1	x	0	-1
C_1	-2	0	-1	-1	-2	0	x	1
D_1	-2	-2	0	-1	-2	-2	1	x

The diagonal elements marked as x's are don't care entries, since these indicate the overlap of an essential sequence with itself, and are not to be considered in the construction of the superstring. Hereafter we will call the superstring constructed by overlapping essential sequences as a supersequence. The entries marked negative indicate that no overlap exists, between the pair of essential sequences under consideration. Their absolute values indicate the length of the required minimal length transfer sequences. The necessity of the negative values arises, because, while two given input sequences always overlap, possibly with an overlap of 0, two SI sequences may not overlap at all. This is indeed the case, when the overlap of the corresponding input sequences is zero and the initial state of one SI sequence is different from the final state of the other SI sequence, or vice versa. When confronted with this problem we resort to a transfer sequence to connect the two SI sequences together.

We now apply the Graph-Reduction Algorithm to the overlap matrix, to obtain the supersequence. We will not explicitly show the reduced graphs at each step as we proceed through the algorithm. We will only show the reduction of the overlap matrix at each step.

Example 3.4 :(continued)

Step 1 :

Select (A_0,C_1), , the max value = 1. Note that the value selected is circled.

	A_0	B_0	C_0	D_0	A_1	B_1	C_1	D_1
A_0	x	-2	0	-1	-2	-2	(1)	-1
B_0	-2	x	-1	-1	-2	0	-1	1
C_0	-2	-2	x	-1	1	-2	0	-1
D_0	-1	-1	-2	x	-1	1	-2	0
A_1	-2	0	-1	-1	x	0	-1	1
B_1	-2	-2	0	-1	1	x	0	-1
C_1	-2	0	-1	-1	-2	0	x	1
D_1	-2	-2	0	-1	-2	-2	1	x

Step 2 : After collapsing (A_0,C_1) into a single vertex we get the following overlap matrix.

	B_0	C_0	D_0	A_1	B_1	A_0C_1	D_1
B_0	x	-1	-1	-2	0	-2	(1)
C_0	-2	x	-1	1	-2	-2	-1
D_0	-1	-2	x	-1	1	-1	0
A_1	0	-1	-1	x	0	-2	1
B_1	-2	0	-1	1	x	-2	-1
A_0C_1	0	-1	-1	-2	0	x	1
D_1	-2	0	-1	-2	-2	-2	x

Select (B_0,D_1) , the max value = 1

Step 3 :

	C_0	D_0	A_1	B_1	A_0C_1	B_0D_1
C_0	x	-1	(1)	-2	-2	-2
D_0	-2	x	-1	1	-1	-1
A_1	-1	-1	x	0	-2	0
B_1	0	-1	1	x	-2	-2
A_0C_1	-1	-1	-2	0	x	0
B_0D_1	0	-1	-2	-2	-2	x

Select (C_0,A_1) , the max value = 1

Step 4 :

	D_0	C_0A_1	B_1	A_0C_1	B_0D_1
D_0	x	-2	①	-1	-1
C_0A_1	-1	x	0	-2	0
B_1	-1	0	x	-2	-2
A_0C_1	-1	-1	0	x	0
B_0D_1	-1	0	-2	-2	x

Select (D_0,B_1) , the max value = 1.

Step 5 :

	C_0A_1	D_0B_1	A_0C_1	B_0D_1
C_0A_1	x	-1	-2	⓪
D_0B_1	0	x	-2	-2
A_0C_1	-1	-1	x	0
B_0D_1	0	-1	-2	x

Select (C_0A_1,B_0D_1) , the max value = 0.

Step 6 :

	D_0B_1	A_0C_1	$C_0A_1B_0D_1$
D_0B_1	x	-2	⓪
A_0C_1	-1	x	-1
$C_0A_1B_0D_1$	-1	-2	x

Select $(D_0B_1, C_0A_1B_0D_1)$ i.e., max value = 0.

Step 7 :

	A_0C_1	$D_0B_1C_0A_1B_0D_1$
A_0C_1	x	⓪
$D_0B_1C_0A_1B_0D_1$	-2	x

Select $(A_0C_1, D_0B_1C_0A_1\ B_0D_1)$, the max value = -1 .

Step 8 :

We get $A_0C_1D_0B_1C_0A_1B_0D_1$

Hence the supersequence $A_0 \bullet C_1 \bullet D_0 \bullet B_1 \bullet C_0 \bullet A_1 \bullet B_0 \bullet D_1$ obtained by overlapping the essential sequences in the order shown in step 8 is the checking experiment for M_1.

The final checking experiment is

$A^0B^0C^1C^0D^1B^1D^0\ A^0B^1D^0A^1C^0D^0A^1\ C^0D^1B^0C^0D^1B^0C^1C$

or in more detail :

input : 0 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1

output : 0 0 1 1 0 0 1 0 0 1 1 1 1 1 1 0 0 1 0 0 1

time : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Note that a transfer sequence of length 1, i.e., T(C,D) is required (since a -1 is selected in step 7) to connect the partial super-sequences $A_0 \bullet C_1$ and $D_0 \bullet B_1 \bullet C_0 \bullet A_1 \bullet B_0 \bullet D_1$. The length of the checking experiment without considering the initialization part is 21. This length can be calculated by knowing the weight wt (H) of the Hamiltonian path constructed by the graph-reduction algorithm. Weight wt (H) equals the arithmetic sum of the maximal overlaps of the sequences in the Hamiltonian path. For our example, wt (H) = 3. Then the formula applies $\lg(s(H)) = lg(ad)^*|Q|^*|A| - wt(H)$

$= 3 * 4 * 2 - 3 = 21$

Initialization to state A is achieved by using either just a homing sequence or a homing sequence followed by a transfer sequence as in example (3.3). In the worst case the total length becomes 25.

It can be easily verified that the final I/O sequence is a checking experiment by trying to obtain the state table from the I/O response. Note that there are 4 distinct responses to the D-sequence 01, thus verifying that M_1 has indeed 4 distinct states. Also the transitions are all implicitly or explicitly verified.

Comparing the length of this sequence to the length found by Hennie's method (example 3.3 we find a significant improvement. Consider another example given in Kohavi [1].

Example 3.5 :

Consider

M_1	0	1
A	B,1	D,0
B	A,0	B,0
C	D,2	A,1
D	D,3	C,1

By inspection 0 is a D-sequence for M_2. The essential sequences are :

$\tilde{\delta}(A,00) = A_0 = A^0B^0A$

$\tilde{\delta}(B,00) = B_0 = B^0A^0B$

$\tilde{\delta}(C,00) = C_0 = C^0D^0D$

$\tilde{\delta}(D,00) = D_0 = D^0D^0D$

$\tilde{\delta}(A,10) = A_1 = A^1D^0D$

$\tilde{\delta}(B,10) = B_1 = B^1B^0A$

$\tilde{\delta}(C,1d) = C_1 = C^1A^0B$

$\tilde{\delta}(D,1d) = D_1 = D^1C^0D$

The overlap matrix of the essential sequences is shown below :

	A_0	B_0	C_0	D_0	A_1	B_1	C_1	D_1
A_0	x	1	-2	-1	0	-1	-2	-1
B_0	1	x	-3	-2	-1	0	-3	-2
C_0	-2	-3	x	1	-2	-3	-1	0
D_0	-2	-3	-1	x	-2	-3	-1	0
A_1	-2	-3	-1	1	x	-3	-1	0
B_1	0	1	-2	-1	0	x	-2	-1
C_1	1	0	-3	-2	-1	0	x	-2
D_1	-2	-3	1	0	-2	-3	-1	x

Applying the graph-reduction algorithm as in example 3.3 we get the following checking experiment without considering the initialization part:

$C^1A^0B^0A^0B^1B^0A^1D^0\ D^1C^0D^0D^0D$

or in more detail :

input : 1 0 0 0 1 0 1 0 1 0 0 0

output : 1 1 0 1 0 0 0 3 1 2 3 3

time : 0 1 2 3 4 5 6 7 8 9 10 11 12

Also we have wt (H) = 4 and

$\lg(s(H)) = lg(ad)^*|Q|^*|A| - wt(H)$

$= 2 * 4 * 2 - 4 = 12$

The checking experiment as obtained by Kohavi [1] without the initialization part is :

input : 0 0 0 1 0 1 0 0 1 0 0 1 1 0

output : 1 0 1 0 0 0 3 3 1 2 3 1 1 1

time : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

$A^0B^0B^0A^0B^1B^0A^1D^0\ D^0D^1C^0D^0D^1C^1A^0B$

and has a length of 14.

Comparing the two results we see that the Supersequence Construction Method yields a better result.

3.8 SUMMARY

In this chapter, checking experiments have been developed by the Supersequence Construction Method. This basically involves forming a SCS of essential sequences [8] using the graph-reduction algorithm. The checking experiments have shorter length compared to those obtained by Hennie's procedure.

In the next chapter an efficient implementation of a special-purpose VLSI chip to find overlaps is presented in detail.

CHAPTER 4

SYSTOLIC DESIGN OF AN OVERLAP CHIP

4.1 INTRODUCTION

Traditionally, the bulk of computer system functionality is implemented in the software medium, as a sequence of instructions for a general-purpose processor. Historically, this has provided the best balance of flexibility, cost, and performance. The new economics of VLSI and continuing advances in VLSI CAD capability opens the possibility of application-specific functionality embedded in silicon as a matter of routine.

Some interesting examples of VLSI designs in traditional software problem domains have appeared in the literature over [30,32] the last few years. Regular language recognizers have been implemented in VLSI [31]. There is no difference in complexity, O(.), between software and silicon implementations of regular language recognizers for either space or time. However O(.) ignores constant factors which are likely to slow down software implementation, while being "free" to the hardware implementation. Organick et al [32] report on a first experiment to transform an Ada program unit into silicon.

Special purpose VLSI chips can function as peripheral devices attached to a conventional host computer. If many types of chips are attached, the resulting system can be considered an efficient general-purpose computer. Figure 4.1 illustrates how special purpose chips such as a pattern matcher, FFT device, and sorter might form part of a general purpose system.

Apart from aiding general-purpose computers, special purpose high-performance computer systems are typically used to meet specific application requirements. As hardware costs continue to drop and processing requirements become well understood, more special purpose systems are being constructed.

In the subsequent section we give a brief review of systolic arrays and their advantages over conventional processor systems. This is followed by a short review of pattern matching chips reported in the literature. The characteristics of the finite input memory machine design for finding overlaps is then compared with the systolic design style. Subsequently we give the systolic design of the overlap chip and illustrate its working by an example. The overlapping of SI sequences (see chapter 3) is then discussed. This is followed by a description of the cell algorithms and cell circuits required to find overlaps between SI sequences. At the end of this Chapter the cell layouts are presented.

4.2 SYSTOLIC ARRAYS

In conventional Von Neumann architectures the processor receives data and instructions from a memory unit and returns the results of its computation to the memory unit. The operation rate that is realizable in such systems is limited by the bandwidth of the processor-memory communication link, commonly referred to as the "Von Neumann" bottleneck [33].

Kung [11] introduced systolic arrays as an elegant and cost-effective architectural solution using VLSI technology to overcome such a bandwidth limitation. A systolic array consists of a collection of processing elements, either all of the same type or a mixture of different types, each capable of performing

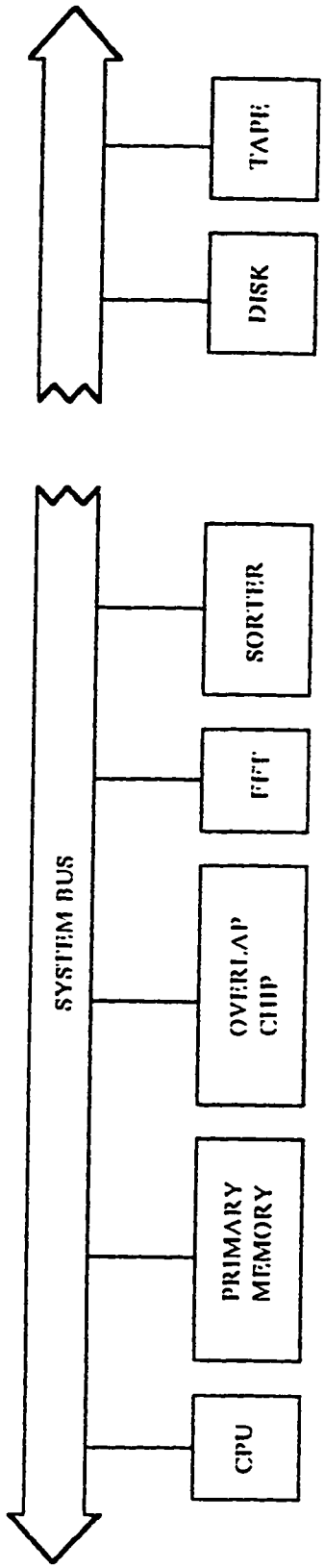


Figure 4.1 Special purpose chips attached to a general purpose computer.

a simple operation such as addition, multiplication, comparison, etc. The processing elements are interconnected to form a linear array, a rectangular mesh, a hexagonal mesh or a tree. Once a data item is retrieved from memory by the processor array, the data item passes through several processing elements in the array before returning to memory. This feature of using a datum from memory many times over without having to store and retrieve intermediate results gives rise to high computation throughput. Figure 4.2 depicts the basic principle of a systolic system. In a typical application, such arrays would be attached as peripheral devices to a host computer which inserts input values into them and extracts output values from them. Simplicity and uniformity of the processing elements in a systolic array and the regularity of their interconnection make them suitable for VLSI implementation at minimal design costs. Many computationally demanding problems, in areas like, signal and image processing, matrix arithmetic, data structures, graph algorithms, language recognition, dynamic programming, require multiple operations to be performed on each data item in a regular and repetitive manner. Systolic arrays are a cost-effective and efficient alternative for handling such problems.

The main aspects of systolic design style [34] can be summarized as follows:

1) Regular structure : The design consists of replications of a few basic cell types. It should be modular and extensible, so that the overall design is a combination of small designs.

2) Synchronous communication: Short, clocked signaling is emphasized and asynchronous, ripple-through logic designs and signal broadcasting are

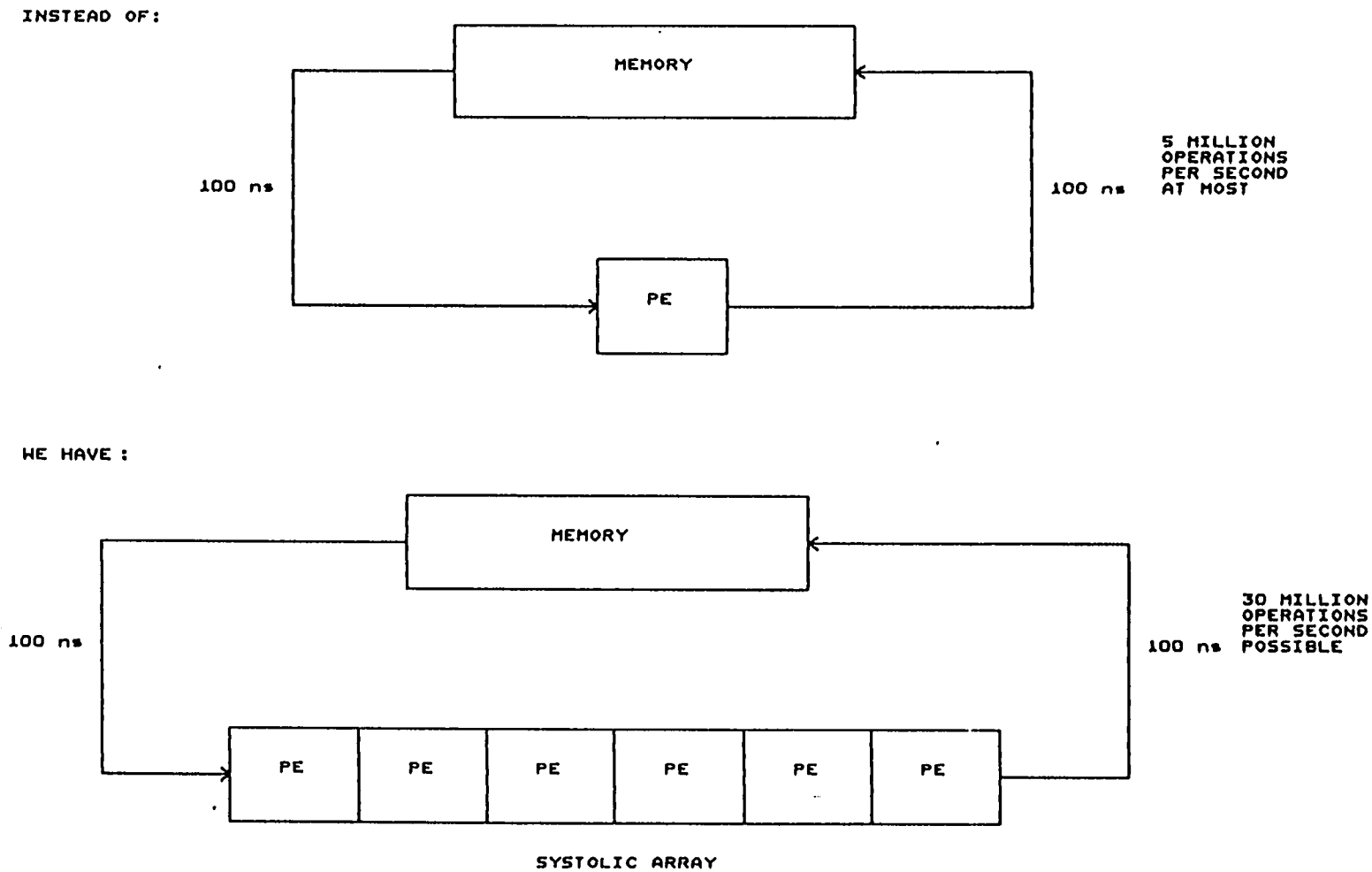


Figure 4.2 Basic principles of a systolic system.

avoided.

3) Pipelining : Pipelining is carried as far as possible, to multiple levels if possible. Typically, several data streams move at constant velocity over fixed paths in the network , interacting at cells where they meet. In this way a large number of cells are active at one time so that the computation speed can keep up with the data rate.

4) Simple Operations : Complex operations are broken down into simple steps, these steps are pipelined, and this is done recursively to lower and lower levels.

5) Nearest neighbour communication : To keep a high data rate, internodal communication is generally restricted to adjacent cells only.

4.3 SYSTOLIC DESIGN OF A VLSI CHIP TO FIND MAXIMAL OVERLAPS

4.3.1 Introduction

In section 2.3.3 we have solved by software the problem of finding the maximal overlap $\psi(s_1,s_2)$ between any two strings s_1 and s_2, where s_1, $s_2 \in \Sigma^+$, by using a modification of the Knuth-Morris-Pratt (KMP) algorithm. In this section we will turn our attention, to the design of a special-purpose VLSI chip to achieve the same result. The systolic implementation which will be described here, like the software implementation also requires $O(lg(s_1))$ time to find the overlap $\psi(s_1,s_2)$ between s_1 and s_2. However O( ) ignores constant factors which

are likely to slow down software implementation, while in the hardware implementation which will be described in this section, we are guaranteed that the maximal overlap $\psi(s_1,s_2)$ is found in $2lg(s_1)-1$ clock pulses.

From a literature search it seems that no chip has been designed to solve the problem of finding maximal overlaps. However, many chips have been designed to solve the classical pattern-matching problem of finding a pattern, P in a text, S. Mukhopadhyay [10] has proposed several machines which store a character of the pattern in each cell and which broadcasts the text string, character by character, to the cells. This broadcasting is the major disadvantage of this algorithm. Each cell requires a connection to the broadcast channel, increasing the power requirements of the system as a whole or decreasing its speed. A chip designed by Mead et al [35] to solve the classical pattern matching problems of finding a pattern in a text, uses another algorithm in which pattern characters are stored in the cells. The text string passes through all of the cells, and the results of character matches are combined using a common-wired-NOR bus. The main disadvantage of this algorithm is the static storage of the pattern and the fact that the cell design is not modular and extensible. However, our preliminary design for the maximal overlap chip consists of a static storage for the pattern, and will be described in the next section. Kung and Foster [11] have designed a pattern matching chip in which both the string and pattern move with a constant velocity in opposite directions. All communication is local, since each cell communicates with its left and right neighbour only. Their design is modular and easily extensible, being based on only a few types of cells. In the systolic design for the maximal overlap chip to be described here, we will follow

Foster and Kung's approach.

4.3.2 Preliminary design:

In section 2.3.4 we have discussed a preliminary design based on a finite input memory machine to find the maximal overlaps. In this section we will see if this design satisfies the conditions for a systolic design as outlined in section 4.2. The basic cell using the finite input memory machine is designed to identify a particular pattern. With a change in the pattern or in its length we cannot just add another cell, but a new cell will have to be designed. Hence a larger chip cannot be obtained by combining smaller basic cells and so it fails to satisfy the first condition. Since the basic cell works in synchronism with a clock and there is no signal broadcasting; it does satisfy the second condition. Pipelining is limited to the fact that the text string can be fed in parallel to the basic cells and all of them will be active simultaneously. But by breaking down the cell function further significant improvement in pipelining can be achieved. Thus the third condition is satisfied to a very limited extent. We see that the cell in fact is doing multiple operations; comparing characters, selecting outputs and storing results. It is possible to break down the cell operation further and pipeline these operations. Thus the fourth condition is not met. The basic cells can only be connected in parallel and cannot be cascaded, hence there is no intercell communication. Therefore the fifth condition does not apply either.

We see from the above discussion that the design based on the finite input memory machine is far from systolic. However it does give us some insight into the problem of finding the overlaps between strings. For a systolic design we

need to make the cell function more distributive, i.e., divide the cell function into simpler steps and perhaps use more than one basic cell. It is along these lines that we will proceed in the next section.

4.3.3 Systolic Design

In Fig. 4.3 the working of the chip we have designed to find the overlaps, is illustrated. The chip, which hereafter we will call the overlap chip accepts two streams of characters from the host machine, and produces a stream of bits which gives us information about the overlap length.

In Fig 4.3 we have considered the set of strings of length m = 4. The string set is $S = \{t_1,t_2,t_3,t_4\} = \{1010,0101,0111,1011\}$ where $t_1,t_2,t_3,t_4 \in \Sigma^*$ and $\Sigma = \{0,1\}$. t_1 is called the P-string (short for pattern string); i.e., the string with which the overlaps with the rest of the strings are to be found. Note that as indicated in Fig 4.3, 3 don't care bits, each denoted by 'x', are pre-catenated with each of t_1, t_2, t_3 and t_4. These are then combined to give the T-string (short for text string). In general if the P-string is of length m, then the number of don't care bits to be pre-catenated with each string in the set S is m-1. The output is a stream of bits, each of which corresponds to one of the bits in the T-string. The data streams move at a steady rate between the host computer and the overlap chip, with a constant time between data items. The interval during which a data item arrives at the chip from either stream is called a beat.

Let us denote the stream of bits in the T-string as $s_1s_2s_3 \ldots s_r$ where $s_i \in \Sigma$, $1 \le i \le r$ and the stream of bits in the P-string as $p_1p_2p_3 \ldots p_m$ where

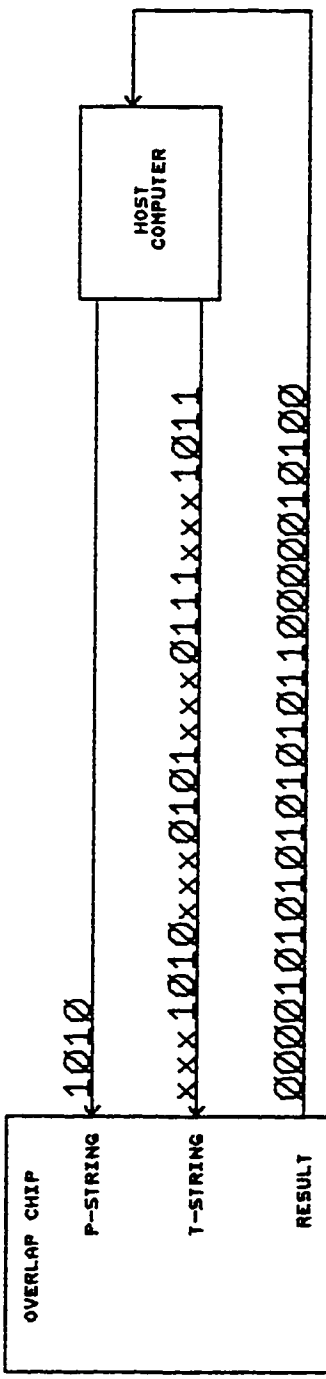


Figure 4.3 Data flow between the host computer and overlap chip.

$p_j \in \Sigma$, $1 \le j \le m$. The length of the T-string is r = n(m + m-1) = n(2m-1), where n is the number of strings in S. The output result stream is denoted as $r_1 r_2 r_3 \ldots r_r$ where $r_k \in \Sigma$, $1 \le k \le r$. The bits in the two input streams are tested for equality, with the don't care bit 'x' deemed to match any letter in Σ. The output bit r_i is to be set to 1 if the substring $s_{i-m} s_{i+1-m} \ldots s_i$ matches the P-string, and 0 otherwise, i.e.,

$$r_i \leftarrow (s_{i-m} \equiv p_1) \wedge (s_{i+1-m} \equiv p_2) \wedge \ldots (s_i \equiv p_m)$$

where $(s_i \equiv p_m) = 1$ if the two bits are equal. For example in Fig 4.3, the P-string 1010 matches the substrings $s_2 s_3 s_4 s_5$, $s_4 s_5 s_6 s_7$, $s_6 s_7 s_8 s_9$, etc ... of the T-string.

Figure 4.4 shows the flow of the three streams and the states of the m processing elements (m = 4 in this case) forming the systolic array in the overlap chip, for 28 beats. The P-string forms a stream moving from left to right in the systolic array and the T-string forms a stream moving from right to left. The result stream also moves from right to left. An underbar is used to indicate the last bit of the P-string. The maximal overlaps $\psi(t_1,t_1)$ and $\psi(t_1,t_2)$ are as indicated.

In order to get the entire overlap matrix we have two alternatives. The first approach is to input a new P-string once the previous P-string has been compared with the entire T-string and recirculate the T-string afresh. A faster approach is to have more copies of the overlap chip and feed different P-strings to each of them but the same T-string. In this way we would get the overlap matrix with one scan of the T-string, there being no need to recirculate the T-string.

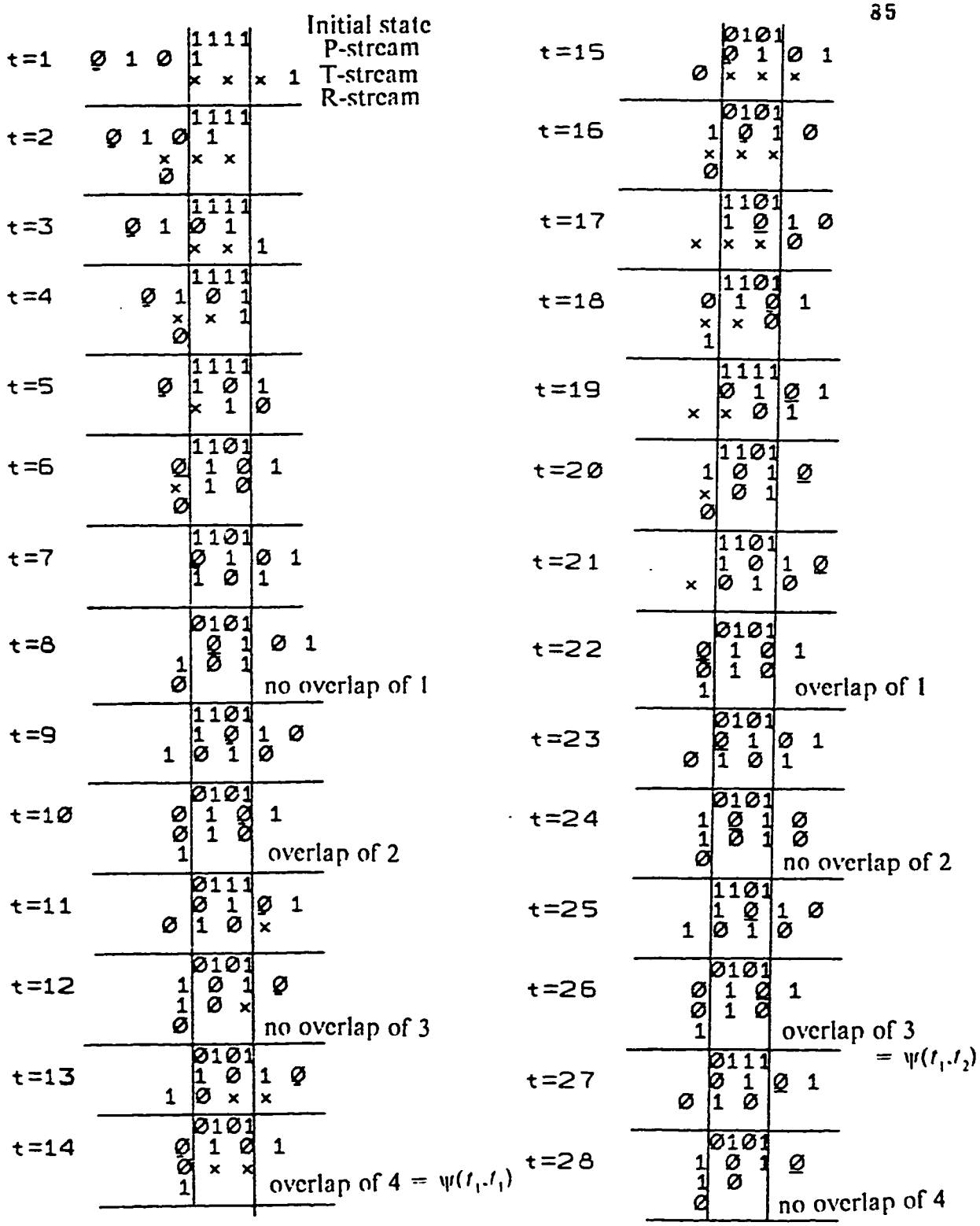


Figure 4.4 Flow of the three streams for 28 beats.

We have illustrated the working of the overlap chip by considering strings over the binary alphabet $\Sigma = \{0,1\}$. However in general we can consider Σ to consist of characters instead of bits. Each character in Σ can be encoded by say, k bits. In particular we denote the wild card character by 'X' and it consists of k don't care bits, each don't care bit being denoted by 'x'.

In the subsequent sections we will discuss the design of the overlap chip in detail.

4.4 ALGORITHM DESIGN - DATA FLOW

We consider here the systolic algorithm involved in our chip design. We will follow a top-down approach in the algorithm design. We will first look broadly at the functions to be performed by the algorithm. Then these functions will be subdivided into simpler ones as we proceed so that the reader gets a true feel of the systolic design style. We begin by examining the data flow algorithm design. Initially we consider the design at the character level. Then as we proceed we will show how to extend this to the bit level.

The character data flow proceeds as follows. The P-string and the T-string arrive alternately over the bus (shown in Fig. 4.3) one character at a time. The interval during which one character arrives from either stream is called a beat. During each pair of consecutive beats the chip must input two characters and output one result. All characters on the chip move during each beat.

Conceptually the overlap chip is divided into character cells, each of which can compare two characters and accumulate a temporary result. On each beat

every character moves to a new cell. The character cells form a linear array in which the P-string and the T-string are moved past each other from opposite directions, and the P-string is re-circulated again and again. The number of character cells is equal to the number of characters in the P-string. A block diagram is given in Fig. 4.5 which illustrates the basic principle. To make each pair of characters of the T-string and P-string meet each other rather than just pass, we separate them by one cell as shown in Fig. 4.5, so that alternate cells are idle. Each cell is active in alternate beats.

The flow of characters through the linear array of 8 cells for several beats is shown in Fig. 4.6. Consider the 6th character cell (indicated by a pointer). Lets trace the history of this character cell for several beats. We see that initially p_1 meets s_i. In the next beat this cell is idle. But during the beat after that it contains p_2 and s_{i+1}. Two beats later p_3 and s_{i+2} are together, then p_4 and s_{i+3} and so on. By the time the last P-string character p_m leaves the cell, the substring $s_i s_{i+1} \ldots . s_{i+m}$, would have met the whole P-string. At each alternate beat the cell compares two characters, one from the P-string and the other from the T-string. Also partial match results are accumulated in the cell, updating of the result being done whenever a new pair of characters enters the cell, and finally the result is output when the last character p_m of the P-string goes past. The results are shifted along with the T-string, so that each match result leaves the array with the last character of its sub-string.

Since each character cell performs two separate functions : (1) comparing characters of the P-string and T-string, and (2) updating and outputting match

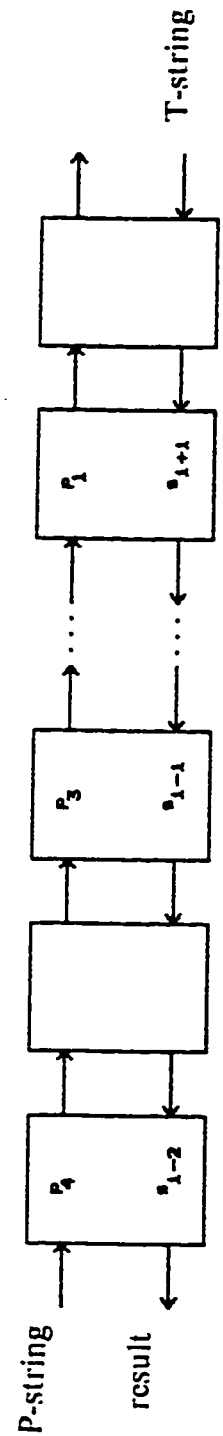


Figure 4.5 Linear array of character cells.

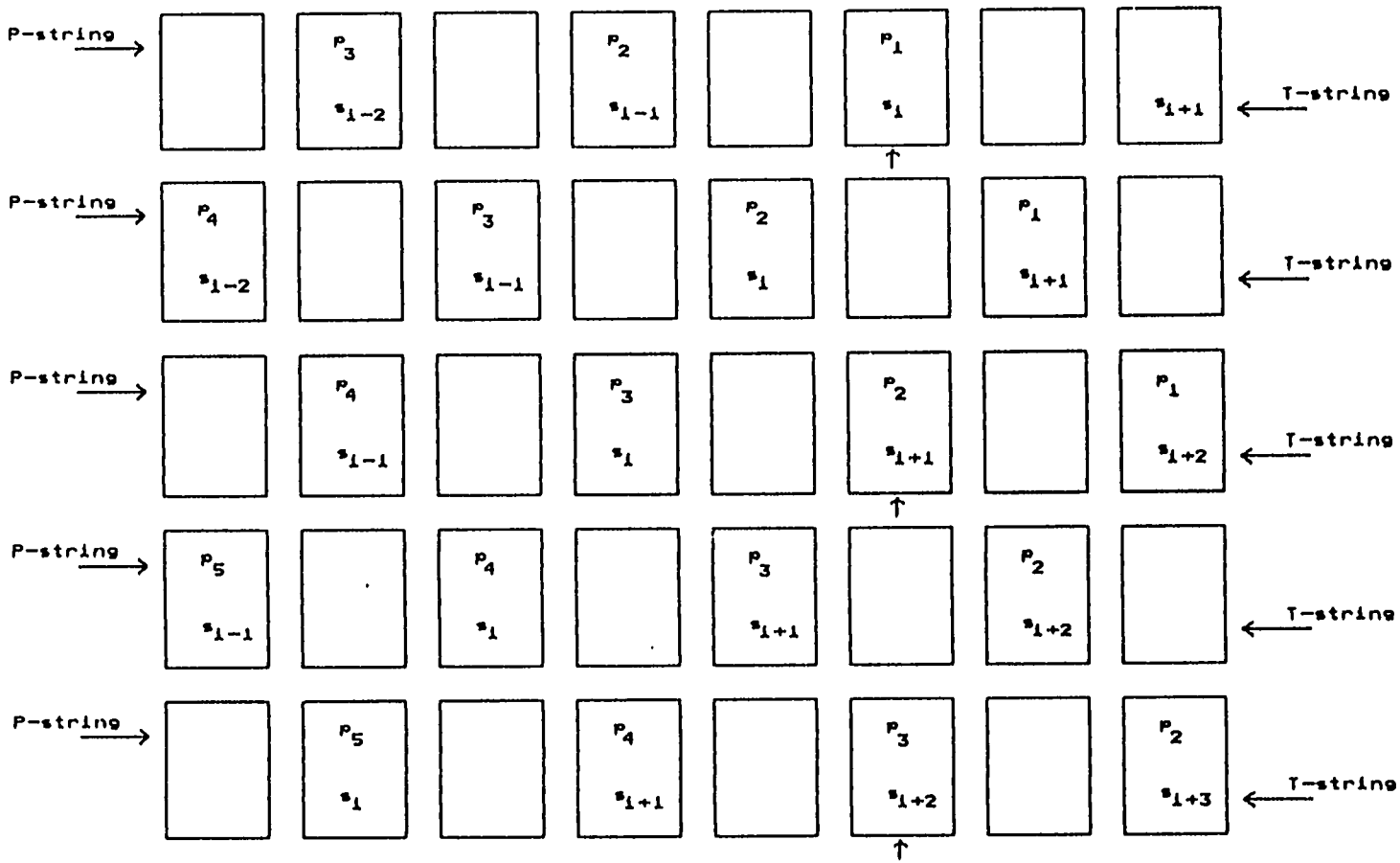


Figure 4.6 Character flow through a linear array of cells.

results, we can divide these functions between two modules. One module does the comparisons and the other module does the updating and outputting of match results. This results in two linear arrays with connections between corresponding cells as shown in Fig. 4.7. The top cells are comparators while the bottom cells will be called accumulators, since they maintain partial results, and shift completed results right to left. Since the T-string contains wild-card characters which match with any character of the P-string, we have an X-stream through the accumulator to mark these bit positions. A '1' in this stream tells the accumulator to ignore the result from the comparator. We also have a λ stream through the accumulators, to indicate the end of the P-string . The λ stream contains a one, if it is the last character of the P-string, and zero, for other characters of the P-string.

The character comparator cell, in general compares two multibit characters. However, the small amount of combinational logic needed to do a bit-parallel character comparison can even be further broken down into a pipelined bit serial comparator as shown in Fig. 4.8. For this second level of pipeline to work, the character bits must be fed into the (now two dimensional) array in staggered formation, so that individual comparison results can accumulate down the comparator cell columns, as shown in Fig. 4.8. The activity pattern in this two-dimensional array now comprises a checker board pattern, reflecting the pipelining going on in both the directions as shown in Fig. 4.9.

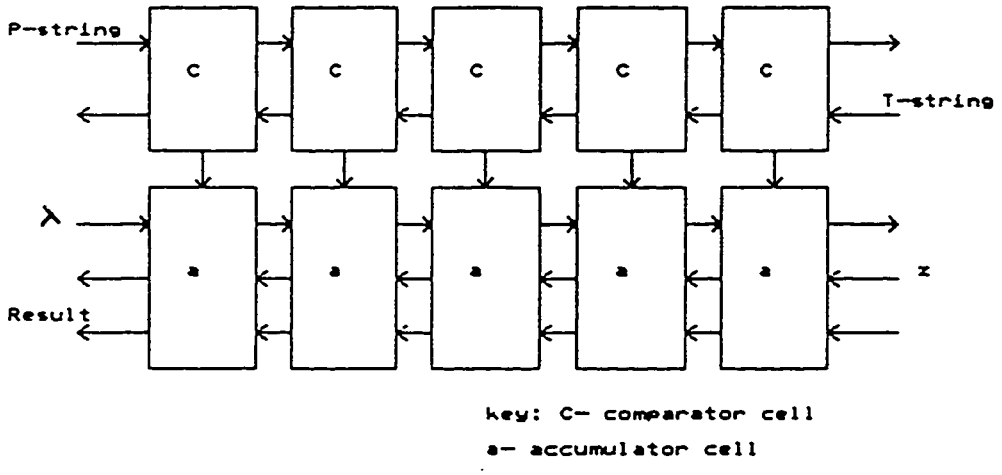


Figure 4.7 Linear array of character cells divided into two modules. Comparators at the top and accumulators at the bottom.

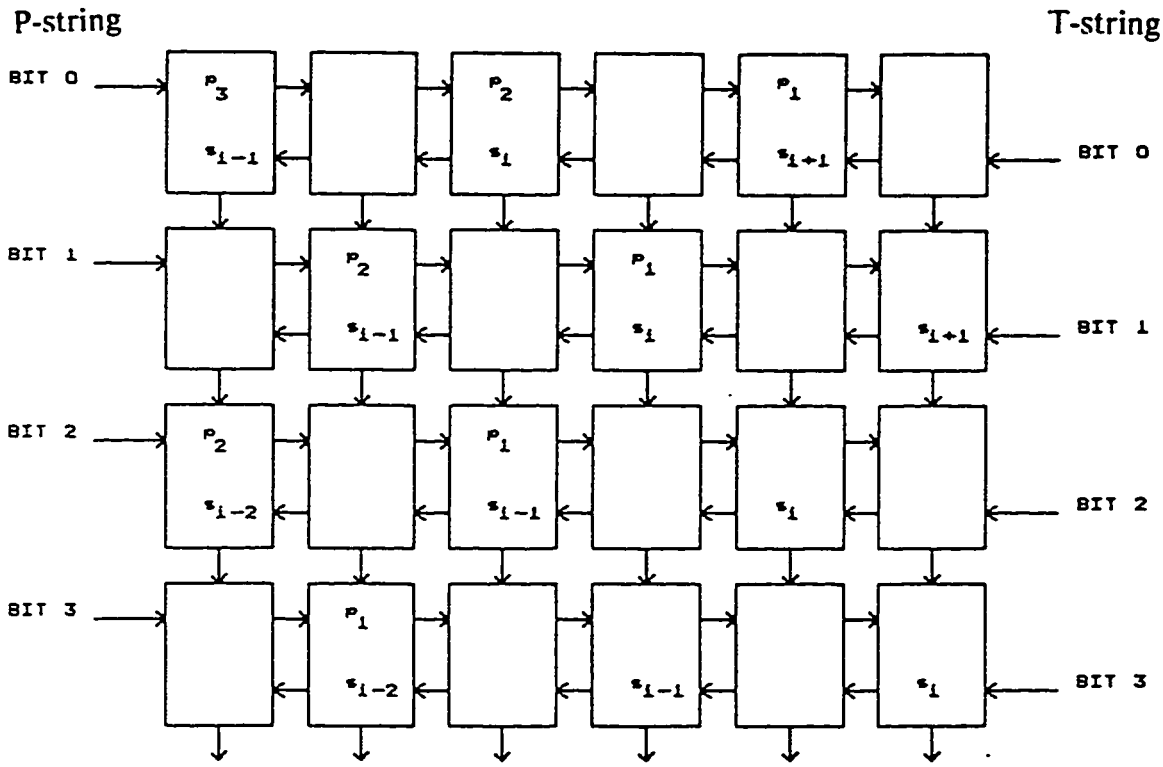


Figure 4.8 Comparator cell (C-cell) breakdown into single-bit comparator cells (c-cell).

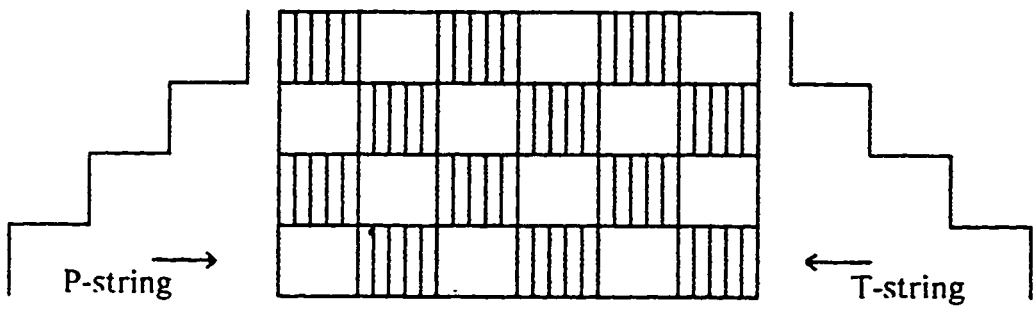


Figure 4.9 Two dimensional cell array with character bits staggered. Active cells from a checkerboard pattern.

4.5 OVERLAPPING OF STATE-INPUT SEQUENCES OR SI SEQUENCES

In section 3.7, we have defined SI sequences. We have also seen that in the Supersequence Construction Method we need to find the overlaps between the essential sequences, which are SI sequences. We will consider an SI sequence to be made up of SI characters. An SI character is defined as a state-input combination. For instance in example 3.4 the essential sequence $A_0 = A^0B^0C^1C^x$ is composed of four SI characters, namely, A^0, B^0, C^1, and C^x. While finding the overlaps between essential sequences, by software we have assumed that a "don't care bit" is associated with the final state, but have not indicated it explicitly. However to find the overlaps by the systolic design algorithm we need to explicitly specify this don't care bit. This don't care bit also necessitates the use of an additional cell called the wild-card cell, apart from the comparator cell and accumulator cell.

The encoding of the SI characters in the essential sequences and the formation of the P-string and the T-string from the essential sequences will now be discussed. Consider the essential sequences of example 3.4. Two bits are necessary to represent the four states, say A = 00, B = 01, C = 10 and D = 11. We need one bit for representing the input associated with each state. Hence each SI character of the essential sequence has 3 bits. The essential sequence A_0 can hence be represented as shown in the following page.

P-string :

$$\tilde{\delta}(A,0d) = A_0 = A^0B^0C^1C$$

$$= \begin{array}{cccc} \emptyset & \emptyset & 1 & 1 \\ \emptyset & 1 & \emptyset & \emptyset \\ \emptyset & \emptyset & 1 & X \end{array} \quad \begin{array}{l} \text{Bit position 2} \\ \text{Bit position 1} \\ \text{Bit position } \emptyset \end{array}$$

We mark the bit positions of the SI characters in the essential sequences as follows. The bit position of the inputs in the essential sequences is 0. The bit position of the encoded states are as follows: The LSB bit is in position 1, the next higher bit is in position 2 and so on. The bit position for the essential sequence A_0 of example 3.4 is as shown above. Note that the bit position 0 has a don't care bit, associated with the final state of the essential sequence. In Fig 4.10 we show the entire schematic, with the 3 different cells which will be required to find the overlaps between the SI sequences. The T-string for the essential sequences in example 3.4 is shown in the following page. It is formed by precatenating 3 wild card characters with each essential sequence and then catenating the resulting sequences.

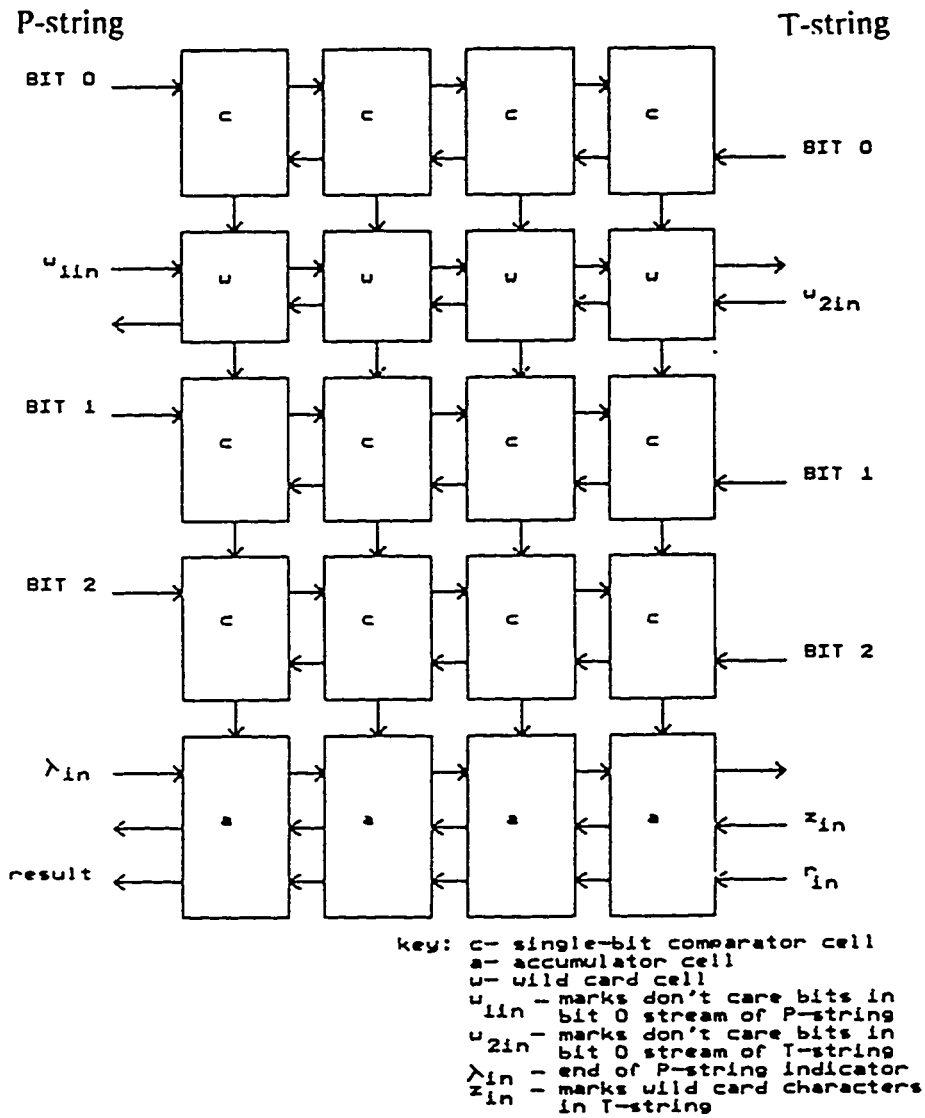


Figure 4.10 Scheme for finding overlaps between SI sequences. (3-bit SI character, 4 character P-string)

T-string :

$$XXXA_0XXXB_0XXXC_0XXXD_0XXXA_1XXXB_1XXXC_1XXXD_1$$

$$= \begin{array}{l} XXXØØ11XXXØ110ØXXX11Ø1XXX1ØØ1XXXØ110ØXXXØ1Ø1XXX1110ØXXX1Ø11 \\ XXXØ1ØØXXX1Ø11XXXØ1ØØXXX1Ø11XXXØØ11XXX1ØØXXXØØ11XXX1ØØ \\ XXXØØ1XXXXØØ1XXXXØØ1XXXXØØ1XXXX1Ø1XXXX1Ø1XXXX1Ø1XXXX1Ø1X \end{array}$$

In general for each essential sequence in the T-string we need to pre-catenate m-1 wild card characters, if the number of SI characters in the essential sequence is m.

4.6 ALGORITHM DESIGN - CELL ALGORITHMS

We need to design three kinds of cells to build an overlap chip, which will find the overlaps between SI sequences. There are the comparator cell, the accumulator cell and the wild card cell. It should be noted that for any other sequence (which is not an SI sequence) the wild card cell may not be needed.

1) One - bit comparator cell

The one-bit comparator has one bit of the P-string flowing from left to right, one bit of the T-string flowing from right to left, and the comparison results for the pair of characters flowing from top to bottom. The one bit comparator cell is shown in Fig. 4.11(a) along with its cell algorithm to update the comparison

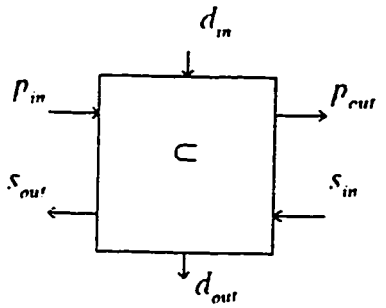
result.

2) Accumulator cell

The accumulator cell and its cell algorithm are shown in Fig. 4.11(c). The accumulator receives d_{in} (the result from the comparator above), λ_{in} (the stream that indicates the end of the P-string), and z_{in} (the stream that marks the wild card characters in the T-string). The λ_{in} stream moves along with the P-stream and indicates by a single '1' the end of the P-string. The z_{in} stream moves along with the T-stream and indicates by a '1' the occurence of a wild card character in the T-string, otherwise it is '0'. The accumulator cell maintains a temporary result t, and when $\lambda_{in} = 1$ uses t to replace the result r that flows from right to left.

3) Wild Card Cell

The wild card cell has two streams, w_{1in} flowing from left to right and w_{2in} flowing from right to left. w_{1in} is used to mark the position of the don't care bit associated with the final state, of the essential sequence forming the P-string. The marking is done as follows: The w_{1in} stream flows in step with the bit 0 stream of the P-string. Whenever a don't care bit appears in the bit 0 stream of the P-string, the w_{1in} stream contains a '1'. For any other bit in the bit 0 stream of the P-string the w_{1in} stream contains a '0'. w_{2in} is used to mark the position of the don't care bit associated with the final state, of the essential sequences forming the T-string. Just like the w_{1in} stream, the w_{2in} stream contains a '1', if

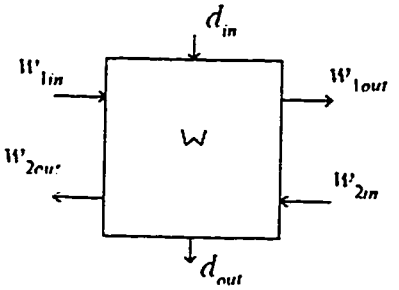


Comparator cell

$p_{out} \leftarrow p_{in}$
$s_{out} \leftarrow s_{in}$
$d_{out} \leftarrow d_{in}\ AND\ (p_{in} = s_{in})$

Cell Algorithm

Figure 4.11(a)

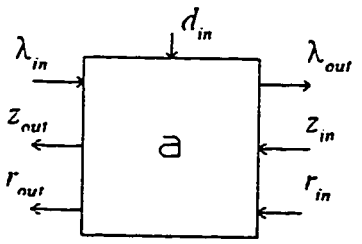


Wild card cell

$W'_{1out} \leftarrow W'_{1in}$
$W'_{2out} \leftarrow W'_{2in}$
$d_{out} \leftarrow (W'_{1in}\ OR\ W'_{2in}\ OR\ d_{in})$

Cell Algorithm

Figure 4.11(b)



Accumulator cell

$\lambda_{out} \leftarrow \lambda_{in}$
$z_{out} \leftarrow z_{in}$
If λ_{in}
then $r_{out} \leftarrow t$, $t \leftarrow true$
else
$r_{out} \leftarrow r_{in}$
$t \leftarrow t\ AND\ (z_{in}\ OR\ d_{in})$

Cell Algorithm

Figure 4.11(c)

there is a corresponding don't care bit in the bit 0 stream of the T-string, else it contains a '0'. The wild card cell along with its cell algorithm is shown in Fig. 4.11(b).

4.7 CIRCUIT AND LAYOUT DESIGN

4.7.1 Data Flow Circuit

Each pipeline used by the algorithm for data flow is implemented as a unidirectional shift register shifting on each beat. Every other cell of the shift register contains valid data.

A shift register is a chain of inverters separated by pass transistors as shown in Fig. 4.12. In our design the inverter has been implemented using CMOS technology. A two phase non-overlapping clock controls the pass transistors. When the gate voltage of the pass transistor is close to the supply voltage V_{dd}, its channel conducts current, while if the voltage is close to ground voltage it does not. The inputs to the inverters can store charges, so data is stored within the inverters. The pass transistors control the inverter inputs. Since adjacent transistors are turned on by opposite phases of the clock, there is never a closed path between inverters that are separated by two pass transistors. Alternate inverters can therefore store independent data bits.

The dynamic alternation of active and idle inverters in the CMOS shift register mirrors the alternations of active and idle cells in the algorithm (compare with Fig. 4.8). Thus each cell can contain one gated inverter from each of the shift registers that passes through it.

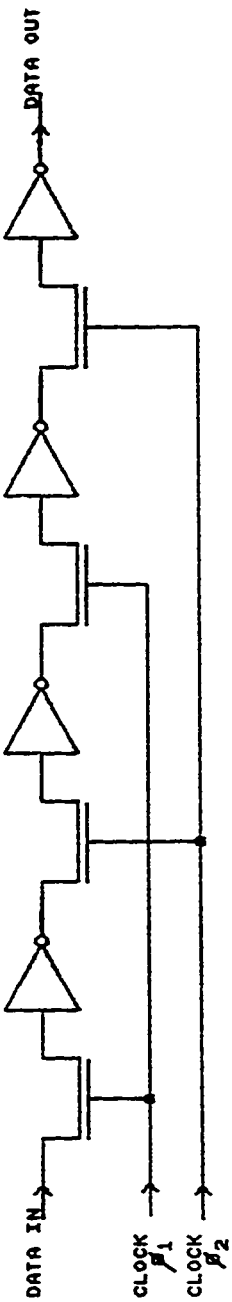


Figure 4.12 A Shift register.

4.7.2 Cell circuit

As described in section 4.7.1, we note that each cell inverts its inputs before sending it to its neighbour. Hence two versions of each of the three cells must be constructed. One version operates on positive inputs to produce inverted outputs, while the other computes positive outputs from inverted inputs. Transforming a cell algorithm to its inverted twin is straight forward. The cell algorithms for the twin versions of the comparator cell, wild card cell and the accumulator cell are shown in Figs. 4.13(a) and (b), Figs. 4.14(a) and (b) and Figs. 4.15(a) and (b) respectively.

The circuit diagrams for the positive and negative comparator cells are shown in Fig.4.16(a) and (b) respectively. When the clock input goes high, the three pass transistors of the comparator cells turn on. (Note that the positive comparator cell is clocked by φ_1 and the negative comparator cell is clocked by φ_2). For the positive comparator cell, the P-string and T-string inputs are then stored on the inverters, and the d - input is stored on one input to the NAND gate. The exclusive NOR gate outputs TRUE if the two inputs are equal, and FALSE otherwise. The output of this equality gate goes to the other input of the NAND gate. If both d_{in} and the output of the exclusive NOR gate are TRUE then $\bar{d}_{out}$ is low, otherwise high.The negative comparator cell is similar, but since it takes inverted inputs and produces positive outputs we need to use an exclusive OR instead of an exclusive NOR, and a NOR gate in place of the NAND gate.

The positive and negative wild-card cells are very much similar to their comparator counterparts. Their circuit diagrams, which are shown in Figs. 4.17(a) and (b) respectively, can be easily deduced from their cell algorithms.

The circuit diagram of the positive and negative accumulator cells are shown in Figs. 4.18(a) and (b) respectively. The pass transistors controlling the inputs to the positive accumulator cell are clocked by φ_1 while for the negative accumulator cell they are clocked by φ_2. For the positive accumulator cell when λ_{in} is TRUE, the assignments $\bar{r}_{out} \leftarrow NOT\ t, t \leftarrow TRUE$ must take place in the correct order (where t is a temporary variable which accumulates the partial result). In order to achieve this we activate the pass transistor controlling $\bar{t}$ when φ_2 is low (i.e., φ_1 is high and cell is active) and set t to be TRUE when φ_2 is high (i.e., φ_1 is low and cell is inactive).

Note that $\bar{r}_{out}$ gets either NOT t or NOT r_{in} depending on whether λ_{in} is TRUE or FALSE. To update the temporary variable we have two NOR gates is cascade as shown in Fig. 4.18(a).

The negative accumulator cell is very much similar. The only changes being that the temporary variable t is initialized to FALSE whenever $\bar{\lambda}_{in}$ is low, and the updating of the temporary variable t is accomplished by two NAND gates in cascade as shown in Fig. 4.18(b).

Finally a floor plan of the systolic overlap chip using the positive and negative versions of the cells is shown in Fig. 4.19.

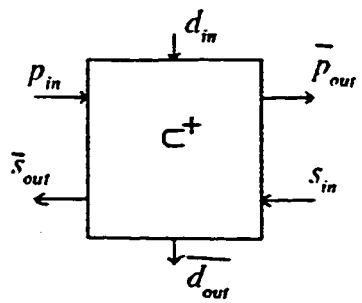
4.8 IMPLEMENTATION OF THE OVERLAP CHIP

The implementation of the three types of cells, both their positive and negative versions have been carried out in CMOS. The layouts have been designed based on the design rules given in [12]. The LUCY layout package available at the graphics centre has been used for this purpose. The layouts of the basic cells are shown in Figs. 4.20 to 4.22.

4.9 SUMMARY

The systolic design of a special-purpose VLSI chip to find overlaps, called the overlap chip has been discussed in detail. The chip design is independent of the pattern. The overlap chip comprises of three basic cells types. For a P-stream of m characters with k bits per character, mk comparator cells, and m wild card and accumulator cells are required. The wild card cell is required when overlapping SI characters only. Two versions of each cell have been used.

The next chaper summarizes the results obtained in this thesis and gives suggestions for future work.



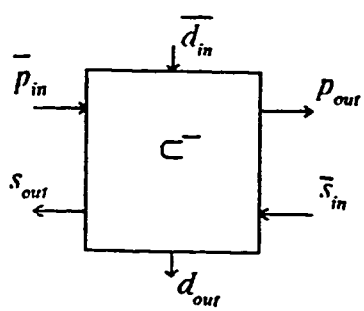
$$\bar{p}_{out} \leftarrow NOT\ p_{in}$$

$$\bar{s}_{out} \leftarrow NOT\ s_{in}$$

$$\overline{d_{out}} \leftarrow d_{in}\ NAND(p_{in} = s_{in})$$

Positive comparator cell **Cell Algorithm**

Figure 4.13(a)



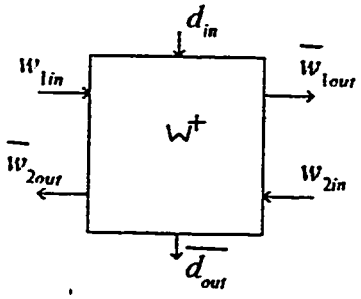
$$p_{out} \leftarrow NOT\ \bar{p}_{in}$$

$$s_{out} \leftarrow NOT\ \bar{s}_{in}$$

$$d_{out} \leftarrow \bar{d}_{in}\ NOR\ (\bar{p}_{in} \neq \bar{s}_{in})$$

Negative comparator cell **Cell Algorithm**

Figure 4.13(b)



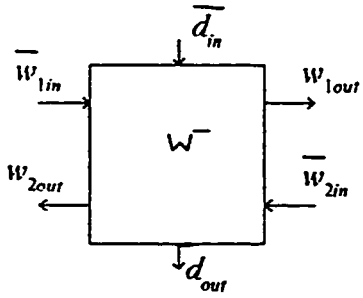
$$\overline{w}_{1out} \leftarrow NOT\ w_{1in}$$

$$\overline{w}_{2out} \leftarrow NOT\ w_{2in}$$

$$\overline{d}_{out} \leftarrow NOT\ (w_{1in}\ OR\ w_{2in}\ OR\ d_{in})$$

Positive wild card cell **Cell Algorithm**

Figure 4.14(a)



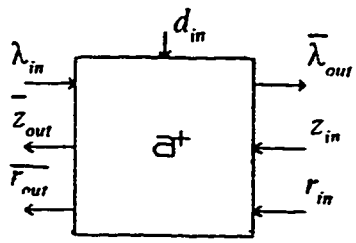
$$w_{1out} \leftarrow NOT\ \overline{w}_{1in}$$

$$w_{2out} \leftarrow NOT\ \overline{w}_{2in}$$

$$d_{out} \leftarrow NOT\ (\overline{w}_{1in}\ AND\ \overline{w}_{2in}\ AND\ \overline{d}_{in})$$

Negative wild card cell **Cell Algorithm**

Figure 4.14(b)



Positive accumulator cell

$\bar{\lambda}_{out} \leftarrow NOT\ \lambda_{in}$

$\bar{z}_{out} \leftarrow NOT\ z_{in}$

If λ_{in}

then $\bar{r}_{out} \leftarrow NOT\ t,\ t \leftarrow true$

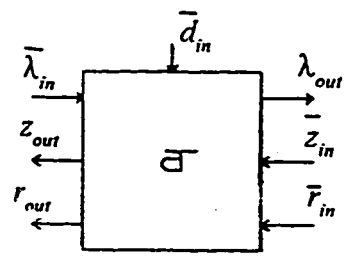
else

$\bar{r}_{out} \leftarrow NOT\ r_{in}$

$t \leftarrow t\ AND\ (z_{in}\ OR\ d_{in})$

Cell Algorithm

Figure 4.15(a)



Negative accumulator cell

$\lambda_{out} \leftarrow NOT\ \bar{\lambda}_{in}$

$z_{out} \leftarrow NOT\ \bar{z}_{in}$

If NOT $\bar{\lambda}_{in}$

then $r_{out} \leftarrow NOT\ t,\ t \leftarrow false$

else

$r_{out} \leftarrow NOT\ \bar{r}_{in}$

$t \leftarrow t\ OR\ (\bar{z}_{in}\ AND\ \bar{d}_{in})$

Cell Algorithm

Figure 4.15(b)

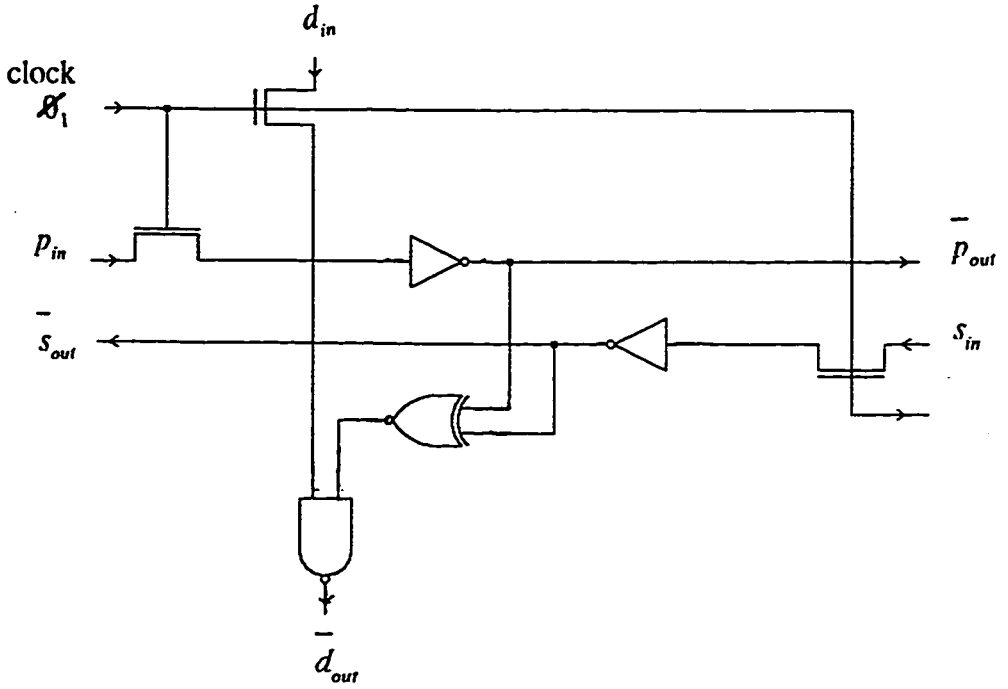


Figure 4.16(a) Positive comparator cell circuit.

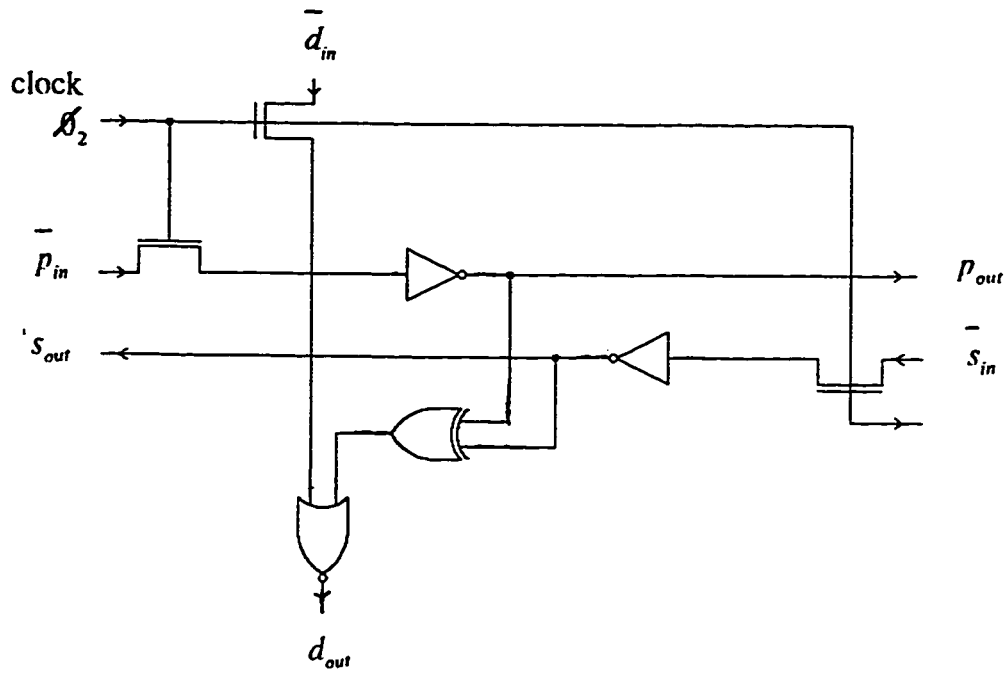


Figure 4.16(b) Negative comparator cell circuit.

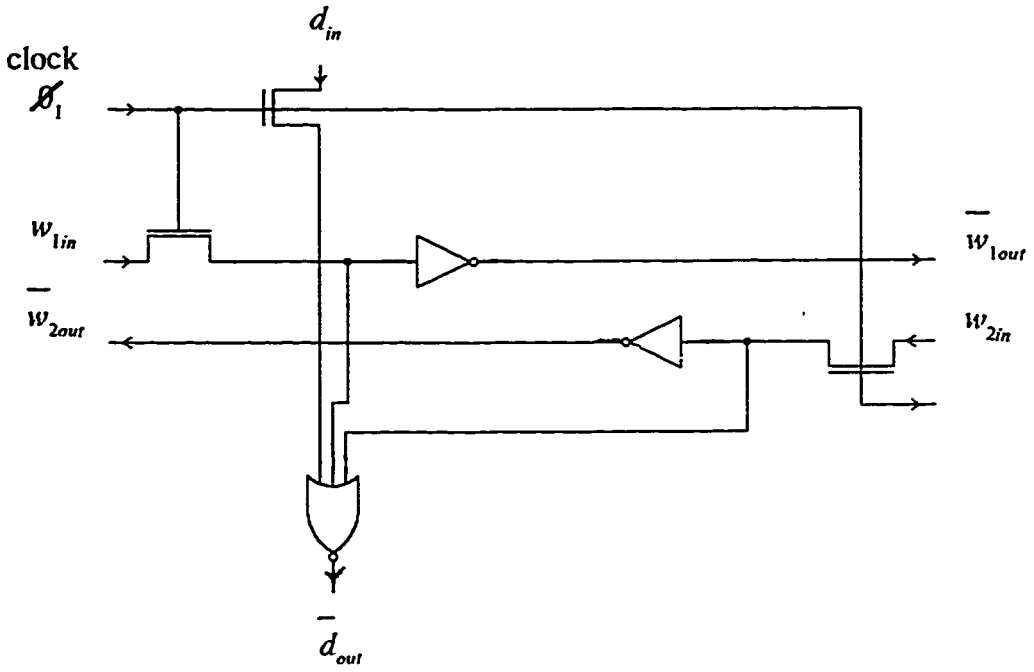


Figure 4.17(a) Positive wild card cell circuit.

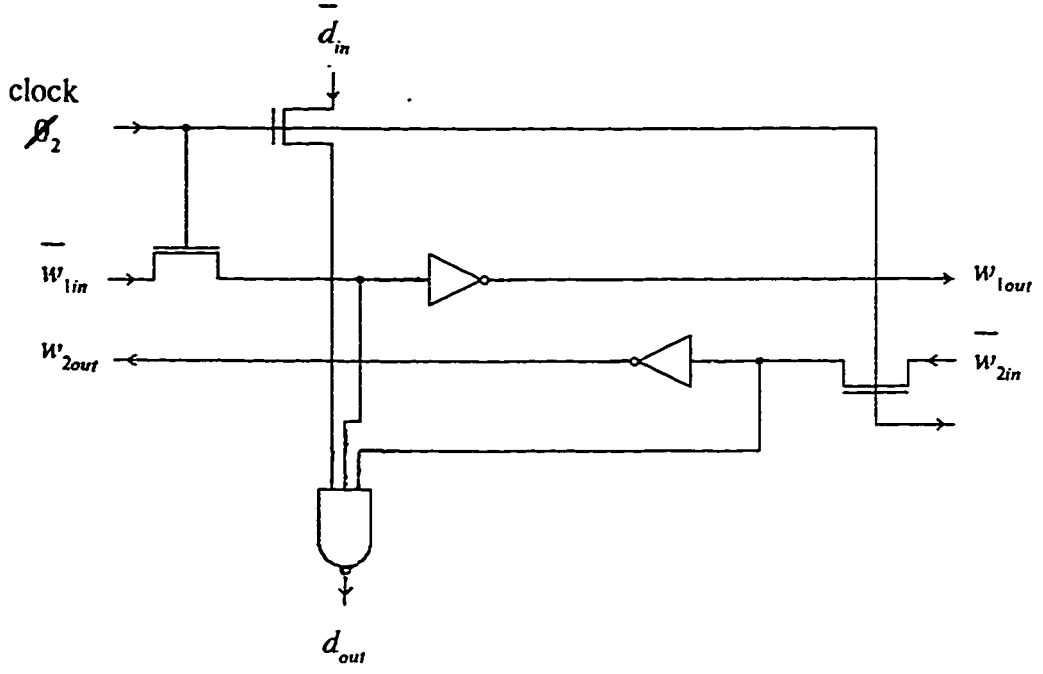


Figure 4.17(b) Negative wild card cell circuit.

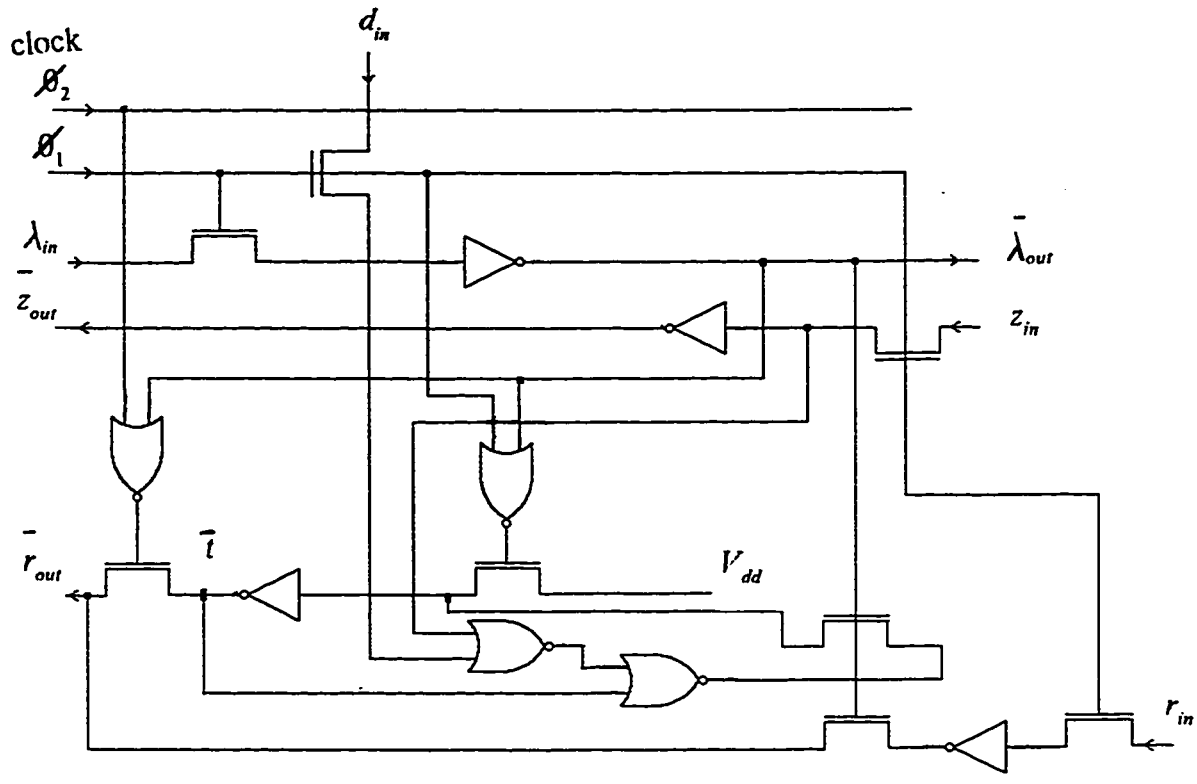


Figure 4.18(a) Positive accumulator cell circuit.

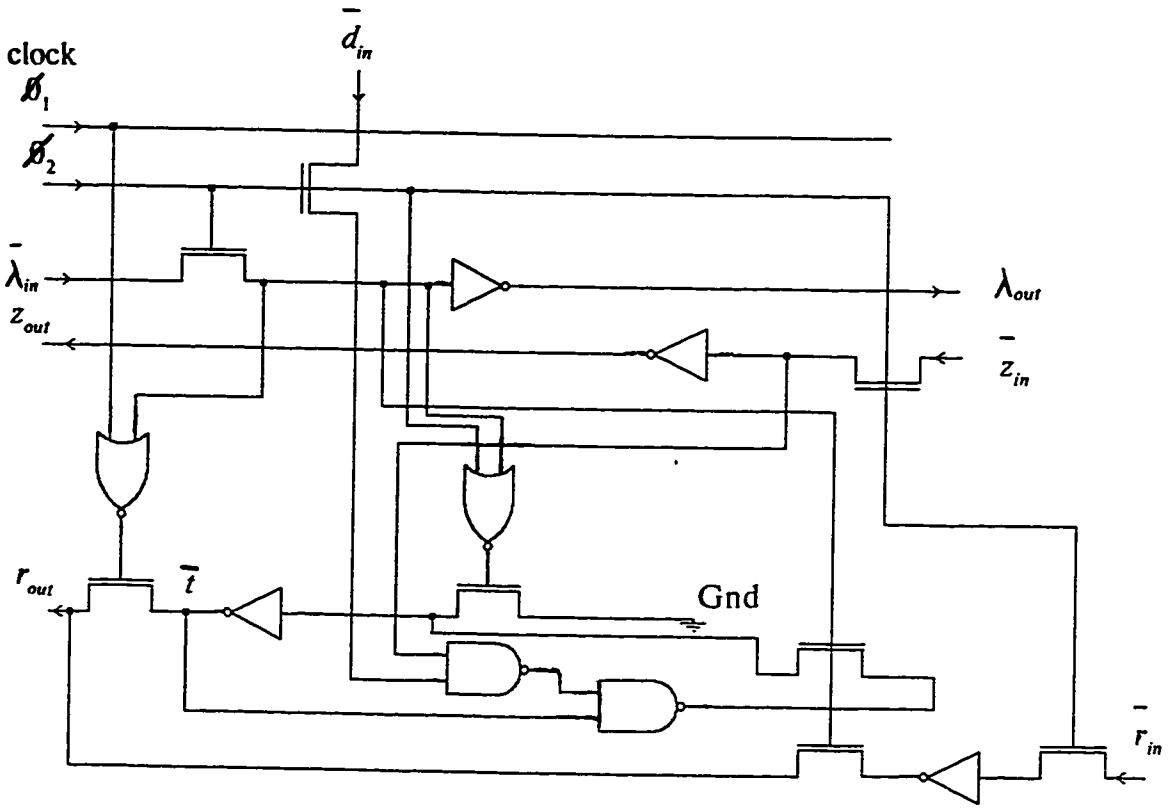


Figure 4.18(b) Negative accumulator cell circuit.

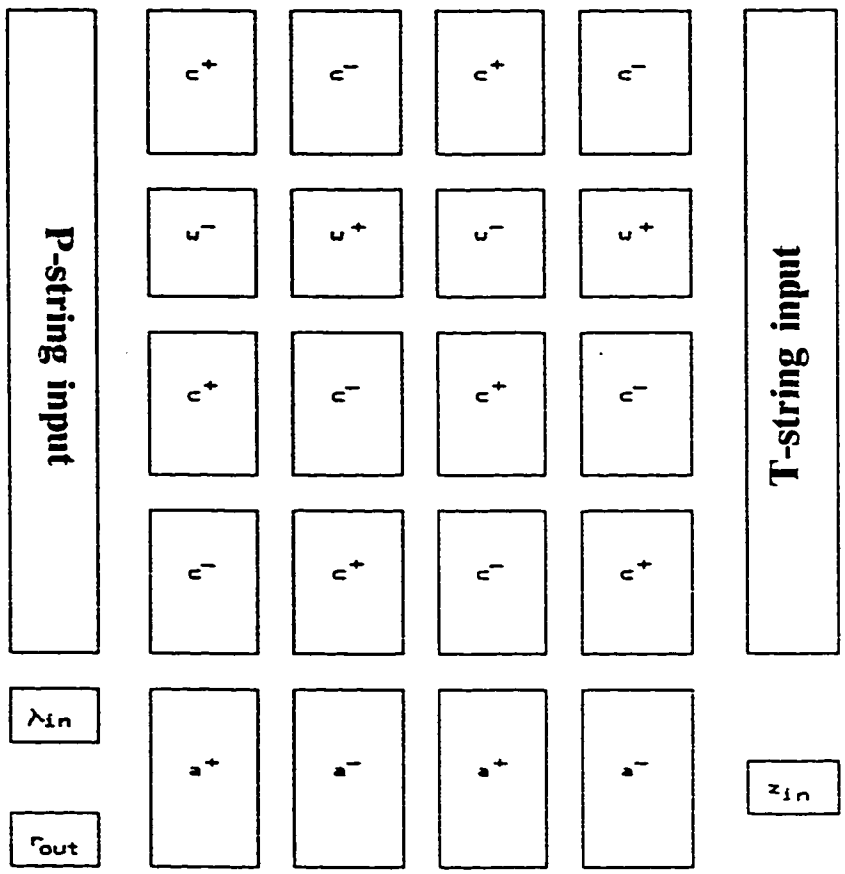


Figure 4.19 Example chip floor plan (3-bit SI character, 4 character, 4 character pattern). For character other than SI w^+ and w^- can be removed.

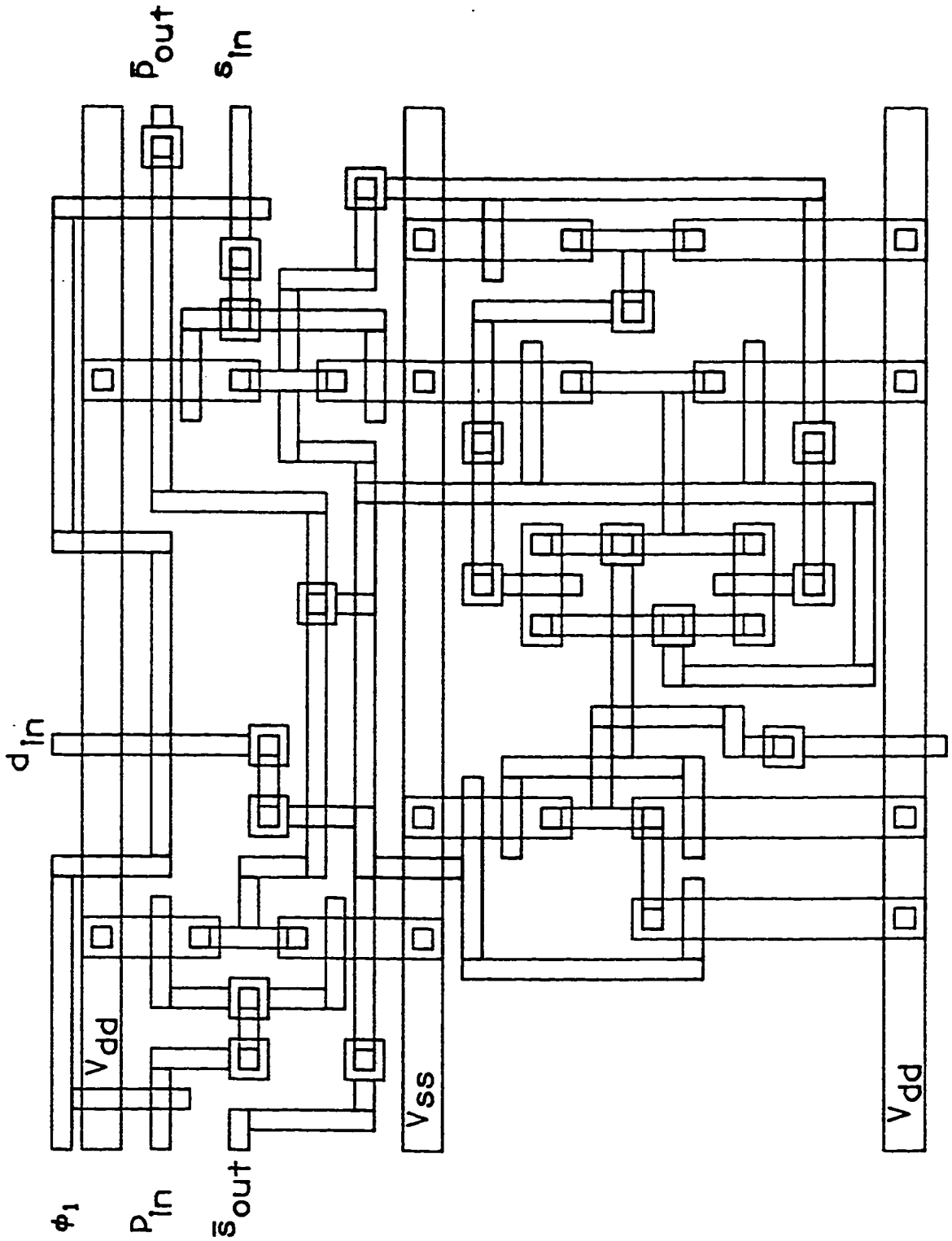


Figure 4.20(a) Positive comparator cell layout.

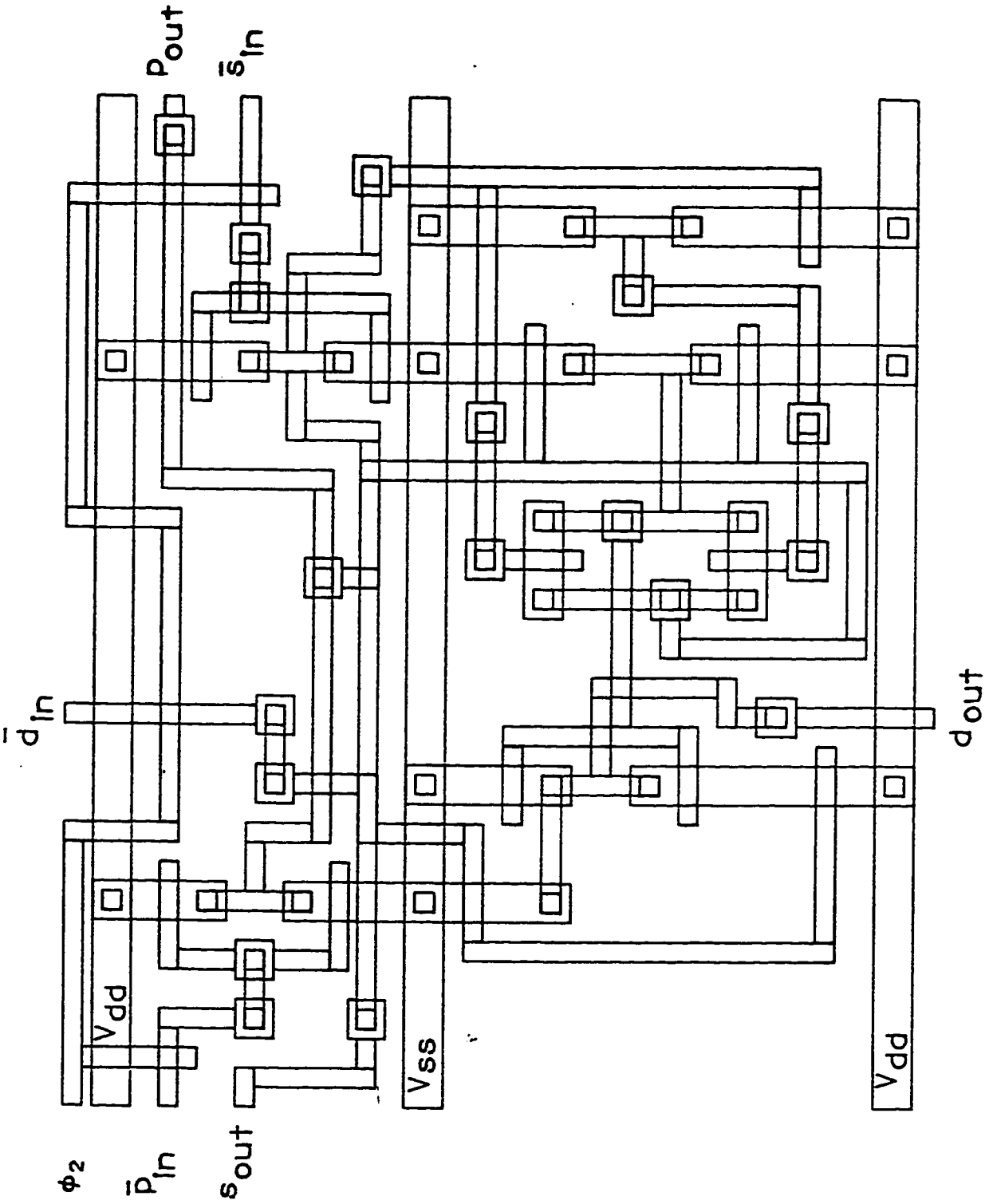


Figure 4.20(b) Negative comparator cell layout.

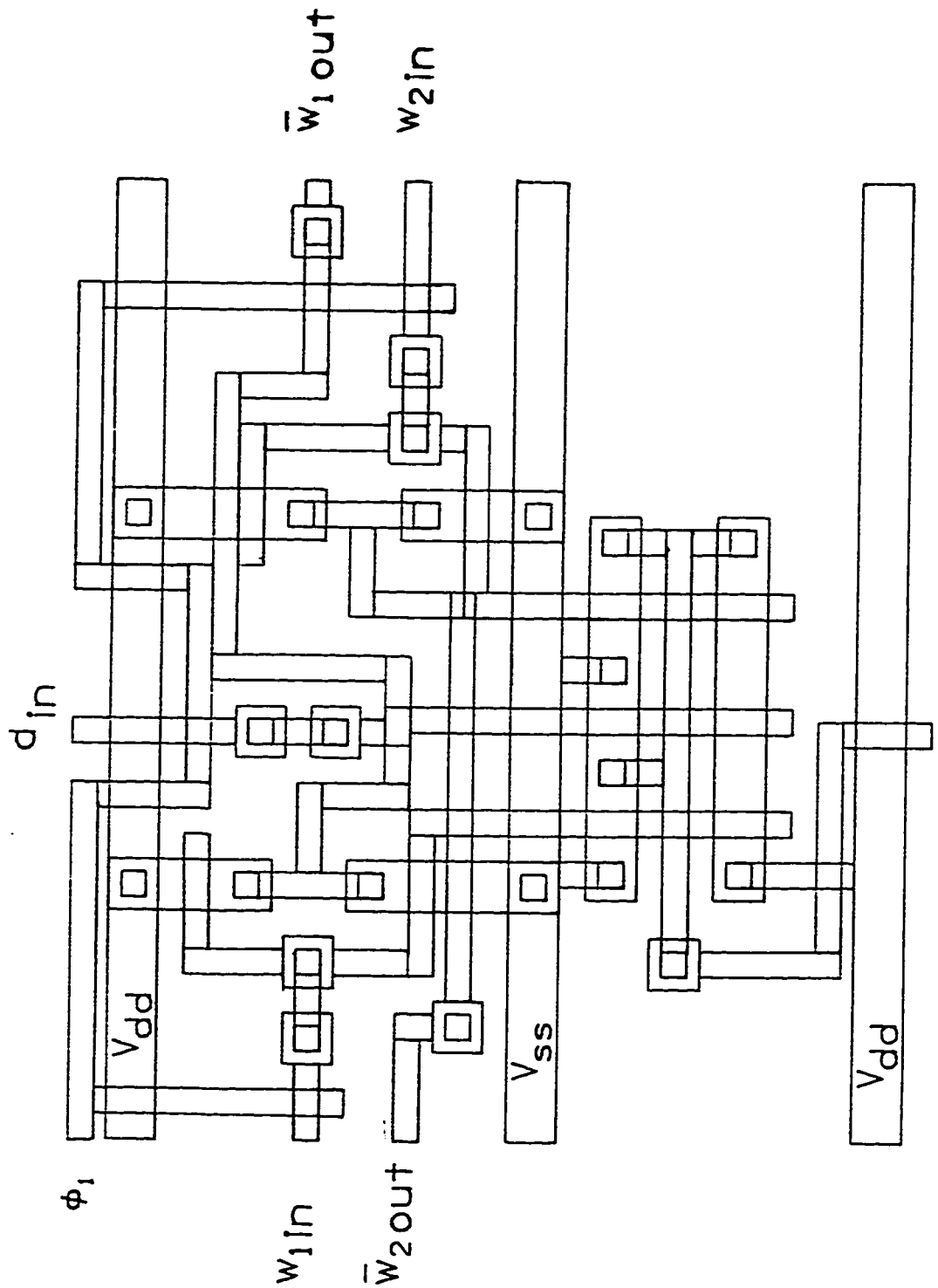


Figure 4.21(a) Positive wild card cell layout.

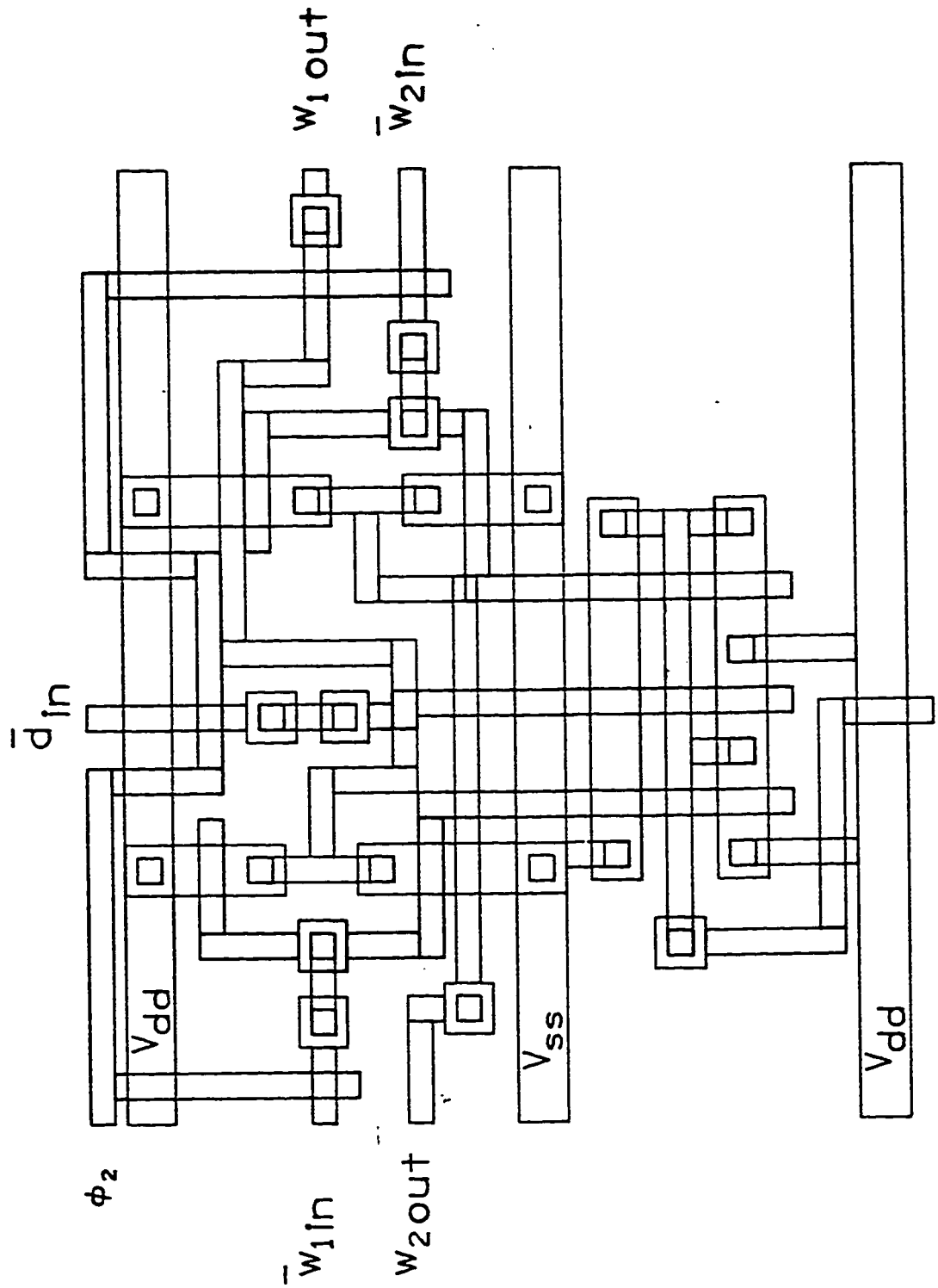


Figure 4.21(b) Negative wild card cell layout.

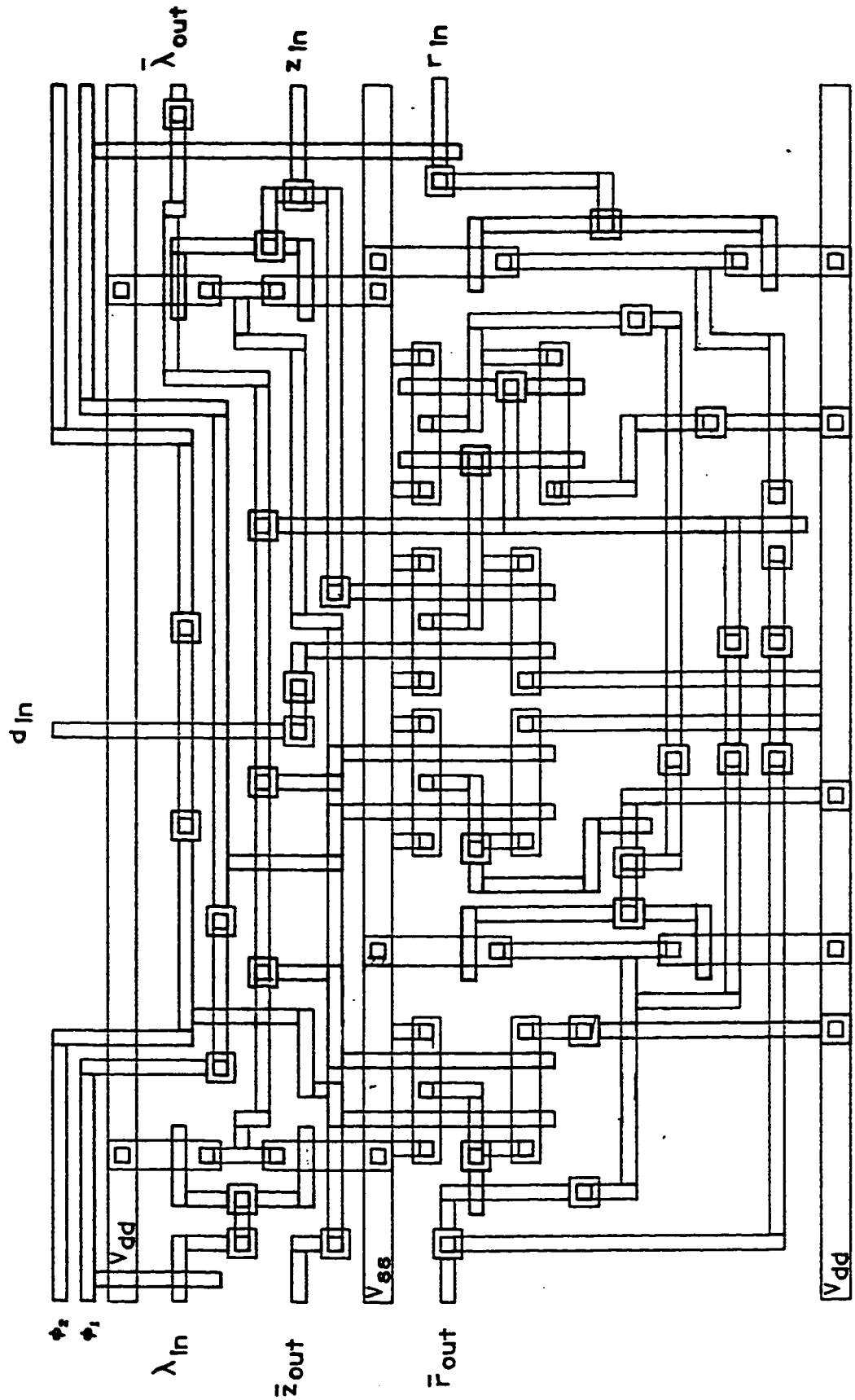


Figure 4.22(a) Positive Accumulator cell layout.

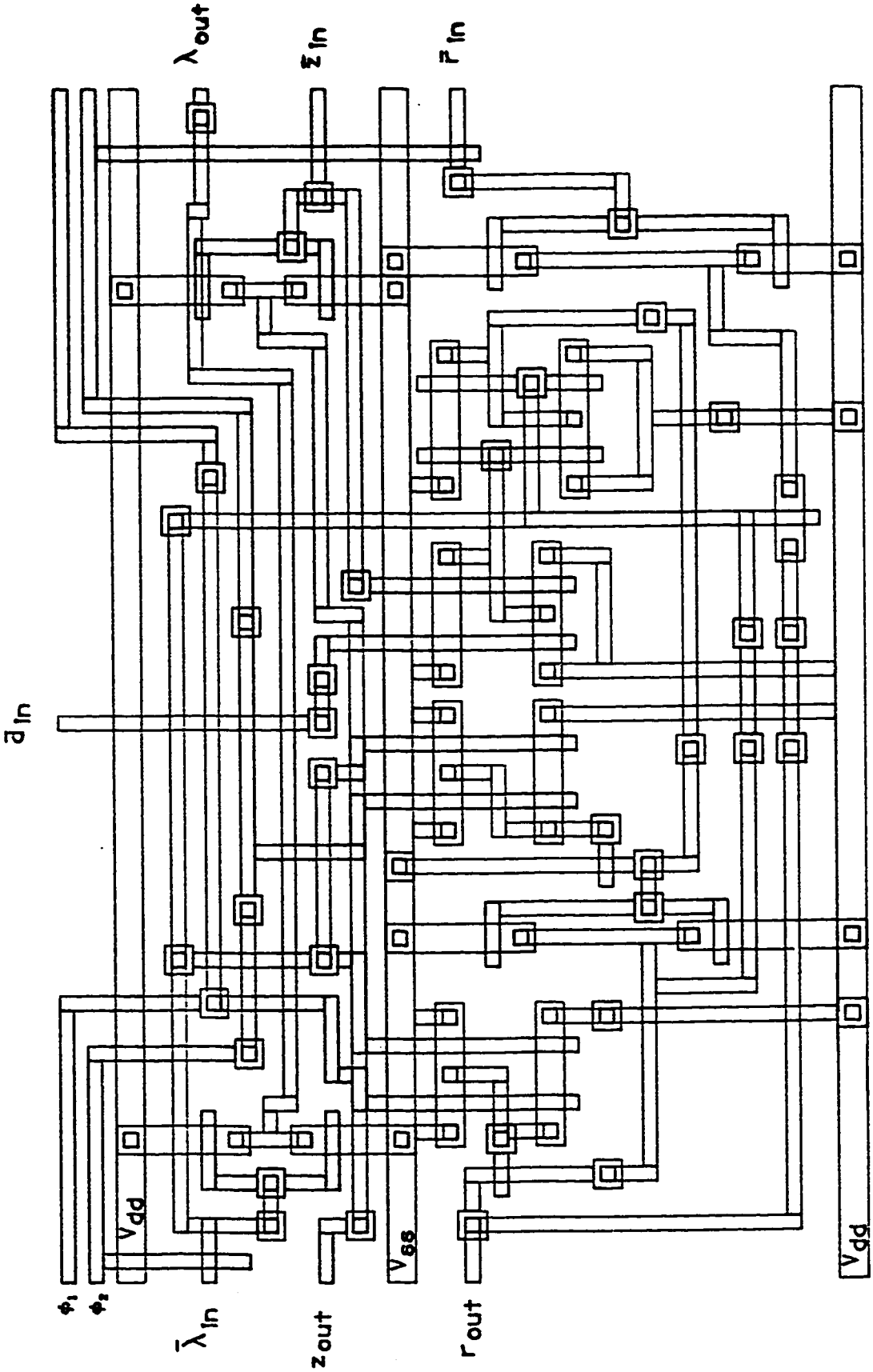


Figure 4.22(b) Negative Accumulator cell layout.

CHAPTER 5
CONCLUSIONS

In this chapter the results obtained in this thesis will be summarized. Furthermore, recommendations for further work along the lines of this thesis work will be given.

5.1 SUMMARY OF RESULTS

The following results have been obtained in this thesis work.

1) The shortest common superstring problem has been investigated. It involves finding the overlapping information and construction of a longest Hamiltonian path in the overlap graph.
2) The maximal overlaps for a given set of strings have been obtained in linear time by using a modification of the Knuth-Morris-Pratt algorithm.
3) Since the shortest common superstring problem is NP-complete, meaning that there is no fast algorithm which can yield an optimal solution, we have investigated several approximation algorithms. These algorithms are based on the construction of a longest Hamiltonian path in the overlap graph obtained from the overlapping information obtained. Of them, the graph-reduction algorithm yields better results in some cases compared to others. In other cases its results are comparable to the greedy or matching algorithm.
4) We have used the graph-reduction algorithm to construct checking experiments, by using essential sequences. The graph-reduction algorithm

constructs a superstring (or supersequence) which along with transfer sequences (used if the essential sequences do not overlap) gives us a checking experiment. The checking experiment has shorter length as compared to conventional methods of obtaining checking experiments.

5) A systolic design of a special purpose VLSI chip, called overlap chip, has been presented in detail. It yields the maximal overlaps between a given set of strings. Though the computational complexity is the same for both the software and hardware implementations, it ignores constant factors which are likely to slow down the software implementation while being free to the hardware implementation.

6) It has been shown how it is possible to overlap SI sequences in hardware, by using a wild card cell in addition to the comparator and accumulator cells.

7) A CMOS implementation of the three basic cells, for both their positive and negative versions has been carried out using the LUCY layout package.

5.2 RECOMMENDATIONS FOR FUTURE WORK

There are several aspects of this thesis work which could provide future research work:

1) Since the problem of finding the shortest common superstring is NP-complete, it would be best to look into heuristic solutions for a particular application. The graph-reduction algorithm lends itself easily for such a task. The characteristics of the overlap matrix for a particular application can be studied and with proper heuristics near optimal solutions can be obtained.

2) Efficient implementations of the algorithms for SCS can be carried out by using proper data structures.

3) We have seen in finding the overlaps, that a hardware implementation is more desirable because of its speed. A method of implementing the software algorithms, for the shortest common superstring in hardware can be studied. This would then yield a total hardware solution to the SCS problem.

APPENDIX - A.1

```
program overlapping;

{ This  program finds the maximal overlaps
  between a given set of strings using a
  modification of the KMP algorithm. }

uses  crt;
const len=4; { max. length of string }
      max=7; { max. no. of strings }
      null= ' ';
type
        pattern = array [1..len] of integer;

var     ov:array[1..max,1..len] of integer;
        temp1,temp2 :pattern;
        jk,i,j,k,l,m:integer;
        ch1,ch:char;
        infile,outf  : text;
        filename :string[20];
        pat1,pat2,flink:pattern;
        found : boolean;

procedure overlap(var x:pattern;y:pattern);

{ Determines the overlap between two patterns,
  x and y using modified KMP. }

var
     ite:integer;
begin
     clrscr;
     flink[1]:=0;
     for ite:= 1 to len do
        begin
          pat1[ite]:=x[ite];
          pat2[ite]:=y[ite];
        end;
     ite:=2;
     while (ite <= len) do
          begin
            jk:= flink[ite-1];
       while (jk<>0) and (pat1[jk] <> pat1[ite-1]) do
                jk:= flink[jk];
            flink[ite]:=jk+1;
            ite:=ite+1;
          end;
     for ite:= 1 to  len do
        found:=false;
     ite:= 1; jk:=1;
     while (ite <= len) and (found = false) do
```

```
          begin
         while ((jk<>0) and (pat1[jk] <> pat2[ite])) do
        begin
           jk:= flink[jk];
        end;
     if (jk <> len) then
        begin
          ite:=ite+1;
          jk:=jk+1;
        end
     else
        begin
          found:= true;
        end;
     end;
     if (found = true) then writeln('success')
       else
          begin
             jk:=jk-1;
          end;
end;

function scanf:char;

{ This function reads a string from an input file. }

begin
   ch:='#';
   read(infile,ch);
 while (ch<>'.') and (ch<>' ') and (eoln(infile)=false) do
        begin
            scanf:=ch;
            read(infile,ch);
        end;
end;

function int(chr:char):integer;

{ Converts text file characters to coressponding integer
  equivalent. }

begin
    int:= ord(chr) - 48;
end;

begin {m a i n}

 { infile  - input file containing the set of strings.
   outfile - output file which finally contains the
   overlaps. }
```

```
    clrscr;
    write('Enter the Input file name : ');
    readln(filename);
    assign(infile,filename);
    reset(infile);

    i:=1;
    while not eof(infile)  do
       begin
         for j:= 1 to len do
            begin
              ch1:=scanf;
              ov[i,j]:=int(ch1);
            end;
         i:=i+1;
         readln(infile);
       end;
    write('Enter the Output file name : ');
    readln(filename);
    assign(outf,filename);
    rewrite(outf);

    for i:=1 to max do
    begin
       for k:=1 to max do
      begin
         for j:=1 to len do
        begin
           temp1[j]:= ov[i,j];
           temp2[j]:= ov[k,j];
        end;
      overlap(temp2,temp1);
      if (k=1) then
         write(outf,jk:1)
      else
         write(outf,jk:2);
    end;
    writeln(outf,'.');
   end;
   close(infile);
   close(outf);
end.
```

APPENDIX - A.2

```
program greedy(infile,outfile);

{ This program finds a SCS using the Greedy
  algorithm. }

uses dos,crt;
const max=132;        { max. no. of edges. }
      maxnode='l';    { max. no. of nodes. }
      null= ' ';

type
    rec=record
         ch1,ch2 :char;
         num :integer;
        end;


var
         data: array[1..max] of rec;
         i,j : integer;
         left,right: array['a'..maxnode] of char;
     leftend,rightend: array ['a'..maxnode] of char;
         u,v,temp1,temp2: char;
         ch1,ch2,num,ch: char;
         infile,outfile: text;
         filename : string[20];


 procedure greedy;

 { Determines the left and right, neighbour
   vertices in the Hamiltonian path. }

 begin
      for u:='a' to maxnode do
      begin
         left[u]:= null;
         right[u]:= null;
         leftend[u]:= u;
         rightend[u]:=u;
     end;
     for i:=1 to max do
    begin
        u:= data[i].ch1;
        v:= data[i].ch2;
        temp1:=leftend[u];
        temp2:=rightend[v];
        if ((right[u] = null) and (left[v] = null)
        and (v<>temp1)) then
        begin
```

```
            right[u]:= v;
            left[v] := u;
            rightend[temp1]:= rightend[v];
            leftend[temp2] := leftend[u];
        end;
    end;
 end;

procedure sort;

{ Sorts the edges according to weight. }

var
    i,j: integer;
    tem: rec;

begin
for i:=1 to max-1  do
    for j:= i+1 to max do
    begin
        if (data[i].num < data[j].num ) then
        begin
        tem:=data[i];
        data[i]:=data[j];
        data[j]:=tem;
        end;
    end;
end;


function scanf:char;

 { Reads a string from the input file. }

begin
     ch:='#';
     read(infile,ch);
     while (ch<>'.') and (ch<>' ')
     and (eoln(infile)=false) do
        begin
            scanf:=ch;
            read(infile,ch);
        end;
end;

function int(chr:char):integer;

{ Gives integer equivalent of character. }

begin
```

```
    int:= ord(chr) - 48;
end;

begin {m a i n}

{ infile  - contains the edges with the
            corresponding weights.
  outfile - gives the right and left
            neighbour vertices in
            the Hamiltonian path. }

     clrscr;
     write('Enter the Input file name : ');
     readln(filename);
     assign(infile,filename);
     reset(infile);


     write('Enter the Output file name : ');
     readln(filename);
     assign(outfile,filename);
     rewrite(outfile);


     i:=1;
     while not eof(infile)  do
     begin
       data[i].ch1:=scanf;
       data[i].ch2:=scanf;
       ch1:=scanf;
       data[i].num:=int(ch1);
       i:=i+1;
       readln(infile);
    end;

    sort;

    greedy;

    for u:='a' to maxnode do
    begin
      writeln(outfile,'left',u,'= ',
      left[u],'  ','right',u,'= ',right[u]);
    end;
    readln;
 end.
```

APPENDIX - A.3

```
program graph_red(infile,outfile);

{ This program finds a longest Hamiltonian path using
  the Graph-Reduction algorithm. }

uses dos,crt;

const max=5;      { max. no. of strings }
      null=' ';

type

      datatype = array[1..max,1..max] of integer;

var
       data: datatype;
       select: array[1..2,1..max] of integer;
       k,d,i,j,e: integer;
       ch1,ch: char;
       infile,outf: text;
       filename: string[20];
       locate,start: boolean;
       path: array[1..max] of integer;


procedure findmax(var data:datatype; m:integer);

{ Finds max. at each step of reduction process. Collapses
  selected edges into a single node. }

var k,i,j, cmax :integer;
    pos         : array[1..2] of integer;
    found       : boolean;
begin
     cmax := -1;
    for i:= 1 to max do
        for j:= 1 to max do
      begin
         if cmax<data[i,j] then
            begin
               cmax := data[i,j];
               pos[1]:=i ;pos[2] := j;
            end;
       end;
   select[1,m] := pos[1];   select[2,m] := pos[2];

   for i := 1 to max do
        begin
           data[pos[1],i] := -1;
           data[i,pos[2]] := -1;
        end;

     found := false;
     if (data[pos[2],pos[1]] <> -1) then
         begin
```

```
                data[pos[2],pos[1]] := -1;
                found := not found
            end;
        k:=2;
        while not found do
          begin
             for i:=1 to m-1 do
                if (pos[k]=select[3-k,i]) then
                  begin
                    pos[k] := select[k,i];
                  end;
                if (data[pos[2],pos[1]]<>-1) then
                  begin
                    data[pos[2],pos[1]] := -1;
                    found := not found
                  end;
                k := 3-k;
          end;
end;

function scanf:char;

{ Reads a string from the input file. }

begin
     ch:='#';
     read(infile,ch);
 while (ch<>' ') and (ch<>'.') and (eoln(infile)=false) do
         begin
             scanf:=ch;
             read(infile,ch);
         end;
end;

function inte(chr:char):integer;

{ Returns corresponding integer value of a character.}

begin
    inte:= ord(chr) - 48;
end;


procedure Hpath;

{ Avoids selecting nodes which have been collapsed into
  a single vertex. Also keeps track of the Hamiltonian
  path. }

begin
    locate:=false; start:=false; i:=1;
    while (start = false) do
        begin
           j:=1;
           while (locate = false) and (j < max) do
```

```
            begin
               if (select[1,i] = select [2,j]) then
                 locate:=true;
               j:=j+1;
            end;
           if (locate=false) then
            begin
            path[1]:=select[1,i];path[2]:=select[2,i];
            start:= true;
            end;
           locate:=false;
           i:=i+1;
      end;

     for j:= 2 to max do
       begin
         locate:=false;
         k:=1;
         while (locate= false) do
           begin
             if (select[1,k]=path[j]) then
               begin
                 path[j+1]:=select[2,k];
                 locate:=true;
               end;
             k:=k+1;
           end;
       end;

  writeln(outf,'The Longest Hamiltonian path is :');

     for i:= 1 to max do
       write(outf,path[i]:3);
     writeln(outf);
end;


begin {m a i n}

{ infile  - input file containing the overlap matrix.
  outfile - output file which gives the final
  Hamiltonian path. }

     clrscr;
     writeln('Enter the Input file name : ');
     readln(filename);
     assign(infile,filename);
     reset(infile);

     write(' Enter the Output file name : ');
     readln(filename);
     assign(outf,filename);
     rewrite(outf);

     i:=1;
```

```
    j:=1;
    while not eof(infile)  do
    begin
      for i := 1 to max do
        begin
          ch1:=scanf;
          data[j,i]:=inte(ch1);
        end;
      readln(infile);
      j:=j+1;
   end;

    for i:= 1 to max do
      begin
         data[i,i]:=-1;
         path[i]:=0;
      end;

    for d:= 1 to max-1 do
      begin
        findmax(data,d);
      end;

    Hpath;

end.
```

APPENDIX - A.4

```
program successor_tree;

{ This program finds a minimal length D-sequence
  using the successor tree method outlined in sec 3.4 }

uses dos,crt;

const
      max=5; { max. no. of states }


type
      datatype = array[1..max,1..5] of char;
      stg = string[30];
      itemptr = ^itemrec;
      itemrec = record
      lp : itemptr;
      rp : itemptr;
      qs : stg;
      fa : itemptr;
      po : byte;
   exist : boolean;
   index : integer;
      end;

var
      i,j,k,cnt1,cnt2,cnt3,len,minim        : integer;
      ch2,ch3,ch1,ch,chr,chr1,chr2,ch4,ch5 : char;
      str,str1,str2,strg,cstr,cstrg         : stg;
      sim,found,fine                        : boolean;
      string1,string2,hstr,hstrg            : stg;
      sstr,sstrg,stg1,stg2                  : stg;
      hmgn,chek,ok,first,rt                 : boolean;
      data                                  : datatype;
      inf                                   : text;
      filename,q,t                          : stg;
      lngth                                 : integer;
      opt,setst,acc0,acc1,acc               : stg;
      min,item,rootnode,tp,son              : itemptr;
      tstr,tstrg,tst                        : stg;
      dseq                                  : string;

procedure print;

begin
   for i:= 1 to max do
      begin
         for j:= 1 to 5 do
         write(data[i,j]:2);
         writeln;
      end;
end;


procedure comp(cstr,cstrg:stg;var sim:boolean);
```

```
{ This procedure compares two strings and tells
  whether they are similar or not .'ABCD' and 'CDBA'
  are similar since both have same components. 'ABCD'
  and 'ABDD' are not similar since 'C' is absent in
  the latter. }

begin
   cstr:=cstr+'#';
   cstrg:=cstrg+'#';
   i:=1;
   sim:=true;
   ch:= cstr[1];
   ch1:=cstrg[1];
   while (ch<>'#') and (sim <> false) do
      begin
         sim:=false;
         j:=1;
         ch1:=cstrg[j];
         while(ch1<>'#') and (sim=false) do
            begin
               if (ch=ch1) then sim:=true;
               j:=j+1;
               ch1:= cstrg[j];
            end;
         i:=i+1;
         ch:=cstr[i];
      end;
   cstrg:=copy(cstrg,1,(length(cstrg)-1));
   cstr:=copy(cstr,1,(length(cstr)-1));
end;

function fnd(str1,str2:stg):boolean;

{ Determines whether the present node matches
  some node in a preceding level. }

var
   sep : boolean;

begin
   str:=''; strg:='';
   cnt1:=1;cnt2:=1;
   found:=false;
   sep:=true;
   str1:=str1+'#';   str2:=str2+'#';
   ch2:=str1[1]; ch3:=str2[1];
   while (ch2<>'#') and (sep=true) do
      begin
         while (ch2<>'0') and (ch2<>'#') do
            begin
               str:=str+str1[cnt1] ;
               cnt1:=cnt1+1;
               ch2:=str1[cnt1];
            end;
```

```
   while (ch3<>'#') and (found=false) do
      begin
         while (ch3<>'0') and (ch3<>'#') do
            begin
               strg:=strg+str2[cnt2];
               cnt2:=cnt2+1;
               ch3:=str2[cnt2];
            end;
         if (length(str)=length(strg)) then
           begin
              comp(str,strg,sep);
              if (sep{sep}=true) then found:=true;
           end
         else  sep:=false;
         cnt2:=cnt2+1;
         if (ch3='0') then ch3:='@';
         strg:='';
      end;
      ch3:=str2[1];
      cnt2:=1;
      str:='';
      found:=false;
      cnt1:=cnt1+1;
      if (ch2='0') then ch2:='@';
    end;
      If (not sep) then fnd:=false;
end;

procedure huncrt(hstrg:stg;var hmgn:boolean);

{ Determines whether a non-trivial component of
  a node vector is homogeneous. }

begin
   hstrg:=hstrg+'#';
   i:=1;  j:=2; k:=2;
   hmgn:=false;
   chr:=hstrg[1];
   chr1:=hstrg[2];
   while (hmgn <> true) and (k <= length(hstrg)-1) do
   begin
      while (chr1<>'#') and (hmgn <> true) do
        begin
           hmgn:=false;
           if (chr=chr1) then hmgn:=true;
           j:=j+1;
           chr1:=hstrg[j];
        end;
      i:=i+1; k:=k+1;
      j:=k;
      chr:=hstrg[i];
      chr1:=hstrg[j];
   end;
   hstrg:=copy(hstrg,1,(length(hstrg)-1));
end;
```

```
function chk(u:stg):boolean;

{ Splits the node vector and uses procedure huncrt
  to check if the node vector is homogeneous. }

var
   p     : integer;
   a,b   : stg;
   flg   : boolean;

begin
   flg:=false;
   repeat
     p:=pos('0',u)-1;
     a:=copy(u,1,p);
     if (p=-1) then a:=u;
     huncrt(a,flg);
     delete(u,1,p+1);
  until ((p=-1) or (flg)) ;
  chk:=flg;
end;


function delt(x:char;k:integer) : char;

var
     i : integer;
     z : char;

begin
   i:=1;
   if (x=data[1,1]) then delt:=data[1,2*k+2];
   while (x<> data[i,1]) do
      i:=i+1;
      delt:=data[i,2*k+2];
      z:=data[i,2*k+2];
   end;

function delts(s:stg;k:integer):stg;

{ Calls procedure delt and determines a successor
  component in the successor tree. }


var
   j : integer;
   t : stg;
   l : integer;
begin
   l:=k;
   t:='';
   for j:= 1 to length(s) do
      begin
```

```
            t:=t+delt(s[j],l);
            delts:=t;
         end;
end;

function lmda(x:char;k:integer) : char;

var
   i : integer;

begin
   i:=1;
   if (x=data[1,1]) then lmda:=data[1,2*k+3];
   while (x<> data[i,1]) do
      i:=i+1;
   lmda:=data[i,2*k+3];
end;

function lmdas(s:stg;m:integer) : stg;

{ Calls procedure lmda and groups components with
  same output. Required in forming the sucessor node. }

var
   j : integer;
   t : stg;
begin
   t:='';
   for j:= 1 to length(s) do
      begin
         t:=t+lmda(s[j],m);
         lmdas:=t;
      end;
end;


function scanf : char;

{ Reads a string from a file. }

begin
   ch:='#';
   read(inf,ch);
   while (ch<>' ') and (ch<>'.') and (eoln(inf)=false) do
     begin
        scanf:=ch;
        read(inf,ch);
     end;
end;


procedure setf(setst,opt:stg;var acc:stg);

{ Determines the overall successor uncertainity vector. }
```

```
var
   accum : array[0..10] of stg;
   j,i   : integer;

begin
   for i:=0 to  10 do
   accum[i]:='';
   acc:='';
   i:=1;
   lngth:=length(setst);
   while (i <= lngth) do
      begin
         k:= ord(setst[i])*(ord(opt[i])-48);
         accum[ord(opt[i])-48]
         :=accum[ord(opt[i])-48]+setst[i];
         i:=i+1;
      end;
    for j:=0 to 10 do
      begin
         if (accum[j]<>'') then acc:=acc+'0'+accum[j];
      end;
    acc:=copy(acc,2,length(acc));
end;


function trmn(tstrg:stg) : boolean;

{ Checks to see if the present node vector is a
  trivial uncertainity vector. }

var
   dumy :boolean;

begin
   tstrg:=tstrg+'0';
   len:=length(tstrg);
   dumy:=true;
   i:=1;
   while (dumy=true) and (2*i <= len) do
      begin
         if (tstrg[2*i] <> '0') then
         dumy:=false;
         i:=i+1;
      end;
   trmn:=dumy;
end;


procedure creitem(qstr:stg;var item:itemptr);

{ Creates a new record each time a node is added
  to the successor tree. }

begin
   new(item);
```

```
   item^.lp:=nil;
   item^.rp:=nil;
   item^.qs:=qstr;
   item^.exist:=false;
end;


procedure newst(u:stg;k:integer;var c:stg);

{ Calls procedure setf and determines the
  successor uncertainity. }

var
   p      : integer;
   a,b    : stg;

begin
    c:='';
    repeat
       p:=pos('0',u)-1  ;
       a:=copy(u,1,p);
       if (p=-1) then a:=u;
       setf(delts(a,k),lmdas(a,k),b);
       c:=c+b+'0';
       delete(u,1,p+1);
    until p=-1;
    c:=copy(c,1,length(c)-1);
end;


function check(a,nd:itemptr) : boolean;

{ Checks the three conditions enumerated in sec 3.4
  to determine if a node is terminal. }

label 20;

begin
   if (nd <> nil) then if  (a=nd)  then goto 20;
   if (not fine)  and (not a^.exist) then
     begin
        if nd<>nil then
          begin
            if (nd^.lp <>nil) then fine:=check(a,nd^.lp);
            if not fine then fine:=check(a,nd^.rp);
            fine:=fine or chk(a^.qs) or trmn(a^.qs)
            or (fnd(a^.qs,nd^.qs) and fnd(nd^.qs,a^.qs));
            if trmn(a^.qs) and (min^.index > a^.index)
             then
                begin
                   min:=a ;
                   minim:=min^.index;
                end;
            if fnd(a^.qs,nd^.qs) and fnd(nd^.qs,a^.qs)
            then nd^.exist:=true;
```

```
            end;
        end;
20:   check:=fine;
end;


procedure create(stn,node:itemptr);

{ Recursively creates the left and right sucessor
  nodes in the tree at each step. }

label 10 ;
begin
   if (first) then goto 10;
     fine:=false;
     if check(stn,rootnode) then
       begin
          exit;
       end;

10:  first:= false;
     newst(stn^.qs,0,stg1);
     creitem(stg1,item);
     node^.lp:=item;
     item^.fa:=node;
     item^.index:=node^.index+1;
     item^.po:=node^.po-6;
     newst(stn^.qs,1,stg2);
       write('_____':item^.po-8,stg1,'________');
     creitem(stg2,item);
     item^.fa:=node;
     item^.index:=node^.index+1;
     item^.po:=node^.po+6;
     node^.rp:=item;
       writeln('_____':10,stg2,'_________');

       writeln(' ':item^.po-28,'|       |',' |       |':25 );
     create(stn^.lp,node^.lp);
     create(stn^.rp,node^.rp);
end;


begin {m a i n}
   clrscr;
   writeln('Enter the Input file name : ');
   readln(filename);
   assign(inf,filename);
   reset(inf);
   i:=1;
   j:=1;
   fine:=false;
   ok:=false;
   while not eof(inf)  do
     begin
        for i := 1 to 5 do
```

```
            data[j,i]:=scanf;
          readln(inf);
          j:=j+1;
      end;
    close(inf);
    first:=true;
    q:='';
    for j:= 1 to max do
       q:=q+data[j,1];
    new(rootnode);
    rootnode^.lp:=nil;
    rootnode^.rp:=nil;
    rootnode^.qs:=q;
    rootnode^.po:=33;
    rootnode^.exist:=false;
    rootnode^.index:=1;
    min^.index:=10000;
    minim:=min^.index;
    writeln('_____':28,q,'_____');
    writeln(' ':22,'|                    |');
    create(rootnode,rootnode);
    if (minim <>10000) then
       begin
          writeln('Min path  length is :',min^.index-1) ;
          tp:=min;
          dseq:='';
          while (tp^.index > 1) do
             begin
                writeln(tp^.index,' :',tp^.qs);
                son:=tp;
                tp:=tp^.fa;
                if (tp^.lp=son) then dseq:='0'+dseq
                else dseq:='1'+dseq;
             end;
          if (tp^.index=1) then
        writeln(tp^.index,' :',tp^.qs);
          writeln('The minimum length D sequence is :',dseq);
    end
    else
         writeln('There is no D sequence for the machine.');
    readln;
end.
```

APPENDIX - A.5

```
program D_tree;

{ This program determines a minimal length D-sequence
  using the partitions method outlined in sec 3.4. }

uses dos,crt;

const
     max=5;
     imax=2;

type
     datatype = array[1..max,1..2*imax+1] of char;
     stg=string[30];
     itemptr=^itemrec;
     itemrec=record
      lp : itemptr;
      rp : itemptr;
      qs : stg;
      fa : itemptr;
      po : byte;
   exist : boolean;
   index : integer;
     end;

var
    i,j,k,lngth,tp1,minim,least : integer;
    ch1,ch                      : char;
    data1                       : datatype;
    data2   : array [1..max,1..imax] of string[10];
    first,fine,yes,debut        : boolean;
    inf,outf                    : text;
    lm0,lm1,lmall,ropt          : stg;
    stg1,stg2,filename,tpq      : stg;
    d,e1,e2,dseq                : stg;
    min,item,rootnode,tp,son    : itemptr;

procedure comp(cstr,cstrg:stg;var sim:boolean);

{ This procedure compares two strings and tells whether
  they are similar or not .'ABCD' and 'CDBA' are similar
  since both have same components. 'ABCD' and 'ABDD' are
  not similar since 'C' is absent in the latter. }

begin
   cstr:=cstr+'#';
   cstrg:=cstrg+'#';
   i:=1;
   sim:=true;
   ch:= cstr[1];
```

```
   ch1:=cstrg[1];
   while (ch<>'#') and (sim <> false) do
      begin
        sim:=false;
        j:=1;
        ch1:=cstrg[j];
        while(ch1<>'#') and (sim=false) do
           begin
              if (ch=ch1) then sim:=true;
              j:=j+1;
              ch1:= cstrg[j];
           end;
        i:=i+1;
        ch:=cstr[i];
      end;
   cstrg:=copy(cstrg,1,(length(cstrg)-1));
   cstr:=copy(cstr,1,(length(cstr)-1));
end;

function fnd(str1,str2:stg):boolean;

{ Determines if the current node matches a node
  at a preceding level of the D-tree. }

var
    found,sep        : boolean;
    str,strg         : stg;
    cnt1,cnt2        : integer;
    ch2,ch3          : char;

begin
   str:=''; strg:='';
   cnt1:=1;cnt2:=1;
   found:=false;
   sep:=true;
   str1:=str1+'#';   str2:=str2+'#';
   ch2:=str1[1]; ch3:=str2[1];
   while (ch2<>'#') and (sep=true) do
     begin
        while (ch2<>'0') and (ch2<>'#') do
           begin
              str:=str+str1[cnt1] ;
              cnt1:=cnt1+1;
              ch2:=str1[cnt1];
           end;
        while (ch3<>'#') and (found=false) do
           begin
              while (ch3<>'0') and (ch3<>'#') do
                 begin
                    strg:=strg+str2[cnt2];
```

```
                    cnt2:=cnt2+1;
                    ch3:=str2[cnt2];
                end;
            if (length(str)=length(strg)) then
                begin
                    comp(str,strg,sep);
                    if (sep=true) then found:=true;
                end
            else sep:=false;
            cnt2:=cnt2+1;
            if (ch3='0') then ch3:='@';
            strg:='';
          end;
          ch3:=str2[1];
          cnt2:=1;
          str:='';
          found:=false;
          cnt1:=cnt1+1;
          if (ch2='0') then ch2:='@';
      end;
      If (not sep) then fnd:=false;
end;


procedure isct(cstr,cstrg:stg;var sct:stg);

{ Finds the intersection between components of a
  partition. It is called by procedure tisct. }

var
   i,j,lngth1,lngth2 : integer;
   sel               : boolean;
   ch,ch1            : char;

begin
   lngth1:=length(cstr);
   lngth2:=length(cstrg);
   i:=1;
   sct:='';
   ch:= cstr[1];
   sel:=false;
   while (i<=lngth1) do
      begin
         j:=1;
         ch1:=cstrg[j];
         while (j<=lngth2) and (sel=false) do
            begin
               if (ch=ch1) then
                 begin
```

```
                    sct:=sct+ch;
                    sel:=true;
                  end;
                j:=j+1;
                ch1:=cstrg[j];
              end;
            i:=i+1;
            ch:=cstr[i];
            sel:=false;
        end;
end;


procedure print;

{ Writes the output to a file. }

begin
  writeln(outf,'The State table of the given machine is :');
  write(outf,' ':2);
  for i:=1 to imax do
     write(outf,i-1:4);
  writeln(outf);
  for i:= 1 to max do
    begin
       for j:= 1 to 2*imax+1 do
          write(outf,data1[i,j]:2);
       writeln(outf);
    end;
end;

procedure ps;

{ Determines the previous-state table from the
  given state table. }

var
   l,j,i          : integer;
   pstg           : stg;
   a              : char;

begin
    for i:=1 to max do
       for k:=1 to imax do
          data2[i,k]:='';
    for k:=1 to imax do
       begin
          for i:=1 to max do
             begin
                l:=1;
```

```
                    j:=1;
                    a:=data1[i,2*k];pstg:='';
                    while (j<=max) do
                      begin
                         if (data1[j,2*k]=a)
                           then  pstg:=pstg+data1[j,1];
                         j:=j+1;
                      end;
                    while (a<>data1[l,1]) do l:=l+1;
                      data2[l,k]:=pstg;
                  end;
          end;
end;

procedure deltg(istg:stg;k:integer;var restg:stg);

{ Determines the next state function under input k. }

var
   i,j    : integer;

begin
   restg:='';
   i:=1;
   while (i<=length(istg)) do
      begin
         j:=1;
         while (istg[i]<>data1[j,1]) do
           j:=j+1;
         restg:=restg+data2[j,k];
         i:=i+1;
      end;
end;


procedure tisct(t1,t2:stg;var tot:stg);

{ Finds the overall intersection between two partitions. }

var
    temp,p,q,r : stg;
    x,y        : integer;

begin
   tot:='';
   repeat
      r:=t2;
      x:=pos('0',t1)-1;
      p:=copy(t1,1,x);
      if (x=-1) then p:=t1;
```

```
      repeat
         y:=pos('0',r)-1;
         q:=copy(r,1,y);
         if (y=-1) then q:=r;
         isct(p,q,temp);
         if (temp<>'') then tot:=tot+temp+'0';
         delete(r,1,y+1);
      until (y=-1);
      y:=1;
      delete(t1,1,x+1);
   until (x=-1);
   tot:=copy(tot,1,(length(tot)-1));
end;

procedure cp(u:stg;m:integer;var v:stg);

{ Splits the given node vector and calls procedure
  deltg to determine the next state function under
  input m. }

var
   p    : integer;
   a,b  : stg;

begin
   v:='';
   repeat
      p:=pos('0',u)-1;
      a:=copy(u,1,p);
      if (p=-1) then a:=u;
      deltg(a,m,b);
      if (b<>'') then v:=v+b+'0';
      delete(u,1,p+1);
   until (p=-1)  ;
   v:=copy(v,1,(length(v)-1));
end;

procedure newst(lmd:stg;v:integer;var rlm:stg);

{ Calls procedures cp and tisct. Determines the overall
  successor node vector in the D-tree. }

var
   tlm:stg;

begin
   if (not debut) then
     begin
        if (v=1) then rlm:=lm0;
        if (v=2) then rlm:=lm1;
```

```
        debut:=true;
      end
   else
      begin
         if (v=1) then
           begin
              cp(lmd,1,tlm) ;
              tisct(lm0,tlm,rlm);
           end;
         if (v=2) then
           begin
              cp(lmd,2,tlm);
              tisct(lm1,tlm,rlm);
           end;
      end;
end;


function trmn(tstrg:stg):boolean;

{ Checks to see if the current node vector is a
  trivial uncertainity vector. }

var
   dumy      : boolean;
   len       : integer;

begin
   tstrg:=tstrg+'0';
   len:=length(tstrg);
   dumy:=true;
   i:=1;
   while (dumy=true) and (2*i <= len) do
      begin
         if (tstrg[2*i] <> '0') then
           dumy:=false;
         i:=i+1;
      end;
   trmn:=dumy;
end;

procedure lm(k:integer; var q:stg);

{ Determines the output function under an input k. }

var
   ln,i                 : integer;
   sum                  : array [0..10] of stg;
   s                    : stg;
```

```
begin
   for i:=0 to  10 do
      sum[i]:='';
   q:='';s:='';
   for i:= 1 to max do
      s:=s+data1[i,2*k+1];
   ln:=length(s);
   i:=1;
   while (i <= ln) do
   begin
      sum[ord(s[i])-48]:=sum[ord(s[i])-48]+data1[i,1];
      i:=i+1;
   end;
   for j:=0 to 10 do
   begin
      if (sum[j]<>'') then q:=q+'0'+sum[j];
   end;
   q:=copy(q,2,length(q));
end;


function scanf:char;

{ Reads a string from an input file. }

begin
   ch:='#';
   read(inf,ch);
   while (ch<>' ') and (ch<>'.') and (eoln(inf)=false) do
      begin
         scanf:=ch;
         read(inf,ch);
      end;
end;

procedure creitem(qstr:stg;var item:itemptr);

{ Creates a new record when called. }

begin
   new(item);
   item^.lp:=nil;
   item^.rp:=nil;
   item^.qs:=qstr;
   item^.exist:=false;
end;


function check(hd,nd:itemptr):boolean;
```

```
{ Checks the two conditions outlined in sec 3.4 for
  the partitions method. }

label 20;

begin
   if (nd <> nil) then if  (hd=nd)  then goto 20;
   if (not fine)  and (not hd^.exist) then
     begin
        if nd<>nil then
          begin
             if (nd^.lp <>nil)
               then  fine:=  check(hd,nd^.lp);
             if not fine
               then fine:=  check(hd,nd^.rp);
             fine:=fine or trmn(hd^.qs)
                        or (fnd(hd^.qs,nd^.qs)
                        and fnd(nd^.qs,hd^.qs));
             if trmn(hd^.qs)
                then if (min^.index > hd^.index)  then
                   begin
                      min:=hd ;
                      minim:=min^.index;
                   end;
             if fnd(hd^.qs,nd^.qs)
                and fnd(nd^.qs,hd^.qs)
                 then nd^.exist:=true;
          end;
     end;
20:  check:=fine;
     if (nd^.exist)
      and (hd^.index < nd^.index) then check:=false;
end;


procedure create(stn,node:itemptr);

{ Builds the D-tree recursively. }

label 10 ;

begin
   if (first) then goto 10;
   fine:=false;
   if check(stn,rootnode) then
     begin
        exit;
     end;
10: first:= false;
```

```
    newst(stn^.qs,1,stg1);
    creitem(stg1,item);
    node^.lp:=item;
    item^.fa:=node;
    item^.index:=node^.index+1;
    item^.po:=node^.po-6;
    newst(stn^.qs,2,stg2);
    creitem(stg2,item);
    item^.fa:=node;
    item^.index:=node^.index+1;
    item^.po:=node^.po+6;
    node^.rp:=item;
    create(stn^.lp,node^.lp);
    create(stn^.rp,node^.rp);
end;


begin {m a i n}
    clrscr;
    writeln('Enter the Input file name : ');
    readln(filename);  { filename := 'a:str.dat';}
    assign(inf,filename);
    reset(inf);
    writeln('Enter the output file name : ');
    readln(filename);
    assign(outf,filename);
    rewrite(outf);
    debut:=false;
    fine :=false;
    first:=true;
    j:=1;
    while not eof(inf)  do
       begin
         for i := 1 to 2*imax+1 do
          data1[j,i]:=scanf;
         readln(inf);
         j:=j+1;
       end;
       close(inf);
       print;
       lm(1,lm0);
       lm(2,lm1);
       ps;
       ropt:='';
       for j:=1 to max do
         ropt:= ropt+data1[j,1];
       new(rootnode);
       rootnode^.lp:=nil;
```

```
        rootnode^.rp:=nil;
        rootnode^.qs:=ropt;
        rootnode^.po:=33;
        rootnode^.exist:=false;
        rootnode^.index:=1;
        new(min);
        min^.index:=10000;
        minim:=min^.index;
        create(rootnode,rootnode);
        if (minim <> 10000) then
           begin
               writeln(outf,'The backward trace  of the D-tree to find
               writeln(outf);
               tp:=min;
               least:=min^.index-1;
               dseq:='';
               tp1:=tp^.index;
               tpq:=tp^.qs;
               while (tp1 > 1) do
                  begin
                     writeln(outf,tp1,' :':3,tpq);
                     son:=tp;
                     tp:=tp^.fa;
                     tp1:=tp^.index;
                     tpq:=tp^.qs;
                     if (tp^.lp=son)
                       then dseq:=dseq+'0'
                     else dseq:=dseq+'1';
                  end;
               if (tp1=1)
                 then writeln(outf,tp1,' :':3,tpq);
               writeln(outf);
    writeln(outf,'The minimum length of the D sequence is :',least);
    writeln(outf,'And the minimum length D sequence is    :',dseq);
    end
    else writeln(outf,'There is no D sequence for the machine.':20);
       close(outf);
    end.
```

REFERENCES

[1] Kohavi, Z., *Switching and Finite Automata Theory* , 2nd Ed., McGraw-Hill Co., New York, 1978.

[2] Aho, A. V., and Corasick, M. J., "Efficient String Matching : An Aid to Bibliographic Search," *Communications of the ACM*, Vol 18, No. 6, June 1975.

[3] Mukherjee, A., *Introduction to NMOS and CMOS VLSI Systems* , Prentice-Hall International Inc., 1986.

[4] Tarhio, J., and Ukkonen, E., "A Greedy Approximation Algorithm for Constructing Shortest Common Superstrings," *Theoretical Computer Science* , pp. 131-145, 1988.

[5] Gallant, J., Maier, D., and Storer, J. A., "On Finding Minimal Length Superstrings," *Journal of Computer and System Sciences* , Vol. 20, pp. 50-58, 1980.

[6] Storer, J. A., and Szymanski, T. G., "Data Compression via Textual Substitution," *J. Assoc. Comput. Mach.*, pp. 928-951, 1982.

[7] Turner, J. S., "Approximation Algorithms for the Shortest Common Superstring Problem," *Information and Computation*, Vol. 83, 1-20, 1989.

[8] Ma, S.C.H., "Fault Diagnosis of a Class of Sequential Machines," *M.S. Thesis, King Fahd University of Petroleum and Minerals*, Jan. 1989.

[9] Knuth, D. E., Morris, J. H., Jr., and Pratt, V. R.,"Fast Pattern Matching in Strings," *Siam J. Computing* , Vol. 6, No. 2, June 1977.

[10] Mukhopadhyay, A., "Hardware Algorithms for Non-numeric Computation," *IEEE Transactions on Computers*, Vol. C-28, No. 6, pp. 384-394, June 1979.

[11] Foster, M. J., and Kung, H. T., "The Design of a Special Purpose VLSI Chip," *IEEE Computer Magazine*, Vol. 13, No. 1, pp. 26-40, Jan 1980.

[12] Eshraghian, E., and Weste, N. H. E., *Principles of CMOS VLSI Design* , Addison Wesley Publishing Company, 1985.

[13] Baase, S., *Computer Algorithms: Introduction to Design and Analysis* , Addison Wesley Publishing Company, California, 1978.

[14] Aho, A. V., Hopcroft and Ullman, J. D., *The Design and Analysis of Computer Algorithms* , Addison-Wesley, Reading, Mass. 1974.

[15] Horowitz, E., and Sartaj, S., *Fundamentals of Computer Algorithms* , 2nd ed., Computer Science Press Inc., 1984.

[16] Poage, J. F., and McCluskey, E. J., "Derivation of Optimal Test Sequence for Sequential Machines," *Proc. 5th Annu. Symp. Switching Circuit Theory and Logical Design*, pp. 121-132, 1964.

[17] Seshu, S., and Freeman, D. N., "The Diagnosis of Asynchronous Sequential Switching Systems," *IRE Transactions on Electron. Comput.*,

Vol. EC-11, pp.459-465, August 1962.

[18] Roth, J. P., *Computer Logic, Testing and Verification*, Computer Science Press, Woodland Hills, California, 1980.

[19] Moore, E. F., "Gedanken Experiments on Sequential Machines," *Automata studies* , Princeton University Press, Princeton, N.J., pp 129-133, 1956.

[20] Hennie, F. C., "Fault-Detecting Experiment for Sequential Circuits," *Proc. Fifth Annual Symposium on Sequential Circuit Theory and Logical Design*, 1964.

[21] Hsieh, E. P., "Checking Experiment for Sequential Machines," *IEEE Transactions on Computers* , Vol. C-20, pp. 1152-1166, October 1971.

[22] Friedman, A. R., and Menon, P. R., "Restricted Checking Sequences for Sequential Machines," *IEEE Transactions on Computers*, Vol. C-22, pp. 397-399, April 1973.

[23] Boute, R. T., "Optimal and Near-optimal Checking Experiments for Output Faults in Sequential Machines," *IEEE Transactions on Computers*, Vol. C-23, pp. 1207-1213, November 1974.

[24] Braun, R. D., and Givone, D. D., "An Improved Algorithm for Deriving Checking Experiments," *IEEE Transactions on Computers*, Vol. C-28, pp. 101-108, February 1979.

[25] Lin, G. K., and Menon, P. R. , "Totally Preset Checking Experiment for Sequential Machines," *IEEE Transactions on Computers*, Vol. C-32, pp. 101-108, February 1983.

[26] Beckhoff, G. F., "The Existence and Length of Synchronizing and Distinguishing Sequences," *Int. J. Electronics* 58(5) , pp. 711-728, May 1985.

[27] Lala, P. K., *Fault Tolerant and Fault Testable Hardware Design* , Prentice Hall, Englewood Cliffs, N. J., 1985.

[28] Das, S. R., Chao, P., Chen, Z., Dai, Y. L., and Das, M. K., "Transition Submatrices in Regular Homing Experiments and Identification of Sequential Machines of a Known Class Using Direct-Sum Transition Matrices," *Computers & Operations Research*, Vol. 14, No. 5, pp. 415-433, 1987.

[29] Friedman, A. D., and Menon, P. R., *Fault Detection in Digital Circuits* , Prentice Hall, Englewood Cliffs, N. J., 1971.

[30] Atallah, M. J. and Kosaraju, S. R., "A Generalized Dictionary Machine for VLSI," *IEEE Transactions on Computers*, Vol. C-34, No. 2, pp. 151-155, Feb 1985.

[31] Foster, M. J., and Kung, H. T., "Recognize Regular Languages with Programmable Building Blocks," *J. Digital System*, Vol. 6, No. 4, pp. 323-332, 1982.

[32] Organick, E. I., Carter, T. M., Maloney, M. P., Davis, A., Hayes, A. B., Klaus, D., Linstrom, G., Nelson, B. E. and Smith K. F., "Transforming an Ada Program Unit to Silicon and Verifying Its Behaviour in an Ada Environment: A First Experiment," *IEEE software* , vol. 1, no. 1, pp 31-48, Jan 1984.

[33] Backus, J., "Can Programming be Liberated from the Von-Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, Vol. 21, pp. 613-614, Aug 1978.

[34] Kung, H. T., "Why Systolic Architectures," *IEEE Computer*, Vol. 15, pp. 37-46, Jan 1980.

[35] Mead, C. A., Pashley, R. D., Britton, L. D., Daimon, Y. T. and Sando, S. F., "128-Bit Multicomparator," *IEEE J. Solid State Circuits*, Vol. SC-11, No. 5, pp. 692-695, October 1976.